T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika eriala

Janek Press

KOLMEMÕÕTMELINE RASTERGRAAFIKA

Magistritöö (40 AP)

Juhendaja: Ain Isotamm, PhD

Autor: …… „…“ mai 2007
Juhendaja: …… „…“ mai 2007

Tartu 2007

Sissejuhatus

Käesolev magistritöö on autori uurimus neljanda põlvkonna laiatarbe graafikariistvara uutest võimalustest ja nende rakendamisest. *OpenGL* laienduste kaudu kasutatakse riistvara uusi võimalusi, mis jõudsid laiema avalikkuse ette 2006. aasta lõpus, mil *Microsoftil* valmis operatsioonisüsteem *Vista* ja *Nvidia* alustas 8800 seeria graafikakaartide tootmist, mis toetasid esimesena *Vista* uut reaalaja graafika programmeerimise rakendusteeki *DirectX 10*. Riistvara jõudluse kasv koos uute tarkvaralahendustega loob tingimused üha arvutusmahukamate ülesannete lahendamiseks reaalajas.

Leidub arvukalt visualiseerimissüsteeme staatilise kolmemõõtmelist andmehulga kuvamiseks ning protseduraalseks animeerimiseks, kuid autori panus on käesoleva töö raames realiseeritud enda nägemus prototüüpsüsteemist personaalarvutil, mis võimaldab lõppkasutajal vahetult lähteandmeid redigeerida tagades pideva tagasiside hetkeseisust.

Töö eesmärgid on järgmised:

- anda ülevaade graafikariistava viimase põlvkonna uutest võimalustest;
- luua käepärane raamistik graafikariistvara kasutavate programmide prototüüpimiseks;
- luua näiteprogramm graafikariistvaras geomeetria topoloogia muutmise demonstreerimiseks, kirjeldada arendusprotsessi;
- realiseerida 3D rasterpiltide manipuleerimise vahend, mis võimaldab:
    - väliseid andmeid laadida ja salvestada,
    - kasutajal saada hea ülevaade pildi sisust reaalajas,
    - demonstreerida uue riistvara tööjõudlust.

Tööle on lisatud laserplaat uurimuse käigus loodud tarkvaraga ning stereopiltide vaatamiseks vajalikud prillid.

1. Graafikariistvara uus generatsioon

Graafikakaardid on viimase kümne aastaga teinud läbi suure arengu alustades lihtsatest lisamoodulitest eraldi mälupuhvriga ekraanipildi hoidmiseks kuni üle saja paralleelselt töötava vektorprotsessoriga süsteemideni, mis on jõudluselt võrreldavad superarvutitega. Järgnevalt on esitatud lühike ülevaade neljanda põlvkonna programmeeritava graafikariistvara uuendustest, mis leiavad kasutamist käesoleva töö kolmandas ja neljandas peatükis kirjeldatud süsteemide poolt. Neljanda põlvkonna all on mõeldud *DirectX 10* täieliku toega süsteeme. Ülevaate koostamisel on kasutatud mitmeid allikaid: [3], [24], [25], [26]. Eestikeelse tehnilise sõnavara kasutamisel on aluseks [4] ning [5].

1.1. Funktsionaalsed uuendused

Neljanda generatsiooni programmeeritavad graafikaprotsessorid (ingl. *graphics processing unit*, *GPU*) realiseerivad mitmeid uuendusi, millest antud töö kontekstis on olulisimad:

- geomeetriaprogrammid (ingl. *geometry shaders*) – uut tüüpi joonistusprogramme rasteriseeritava geomeetria topoloogia muutmiseks. Geomeetriaprogrammi sisendiks on eelnevalt käivitatud punktiprogrammi väljund, millele lisatakse vajadusel infot madalaima taseme topoloogia kohta nagu näiteks antud kolmnurga kõrval asetsevate kolmnurkade koordinaadid. Geomeetriaprogrammi sisendiks võivad olla punktid, jooned, joonte järjend, kolmnurgad, kolmnurkade järjend. Väljundi formaat on sisendist sõltumatu; võimalik on genereerida punkte, joonte ja kolmnurkade järjendeid;
- unifitseeritud joonistusprogrammid (ingl. *unified shaders*) – seoses geomeetriaprogrammi tüübi lisandumisega, lähtuvalt senisest riistvara maksimaalset kasutust takistanud eri tüüpi joonistusprogrammide tarbeks spetsiaalse riistvara olemasolust, on uues arhitektuuris realiseeritud üldised arvutusüksused, mis vastavalt vajadusele suudavad töödelda kõiki kolme tüüpi joonistusprogramme;
- pikad joonistusprogrammid – joonistusprogrammi käskude arv on praktiliselt piiratud vaid jõudluse ning kogu pildi loomiseks kulutatava ajaga;
- 32-bitiliste ujukomaarvuliste tekstuuriformaatide filtreerimine – spetsiaalse riistvara kasutamine filtreerimisel tõstab oluliselt arvutuskiirust, võimaldades kõrgemat pildi kvaliteeti;

- ristkülikukujulised (ingl. *non power of two*) tekstuurid – täielikult on loobutud nõudest, et tekstuuride mõõtmed peavad olema arvu kaks astmed;
- kaheksasse väljundpuhvrisse (ingl. *render targets*) korraga kirjutamine.

1.2. Arvutusjõudlus

Käesoleva töö praktilist osa teostades olid neljanda põlvkonna graafikariistvarast kätte saadavad *G80* arhitektuuri realiseerivad *Nvidia 8800* seeria kaardid. *G80* keskprotsessoris on 681 miljonit transistorit, mis on oluliselt rohkem kui hiljuti turule tulnud Inteli *Core 2 Quad* neljatuumalises protsessoris. Unifitseeritud joonistusprogrammide arvutusüksusi on *8800GTX* kaardil 128, *8800GTS* kaardil 96.

G80 arhitektuuris on oluliselt parandatud dünaamilise hargnemise jõudlust, mis võimaldab graafikariistvara kasutada laiaotstarbelisemalt (ingl. *General-Purpose Computation Using Graphics Hardware*, *GPGPU* [16]).

Uue arhitektuuri paindlikkuse ning riistvara jõudluse demonstreerimiseks on antud töö raames välja töötatud raamistik, millel baseeruvalt on loodud uuendusi kasutavad süsteemid, mida seni ei olnud võimalik graafikariistvaral realiseerida arvutusmahukuse või funktsionaalsuse puudumise tõttu.

2. *OpenGL* raamistik

Käesoleva magistritöö ühe osana loodi programmeerimiskeeles *C++* raamistik *Microsoft Windows* operatioonisüsteemidele, eesmärgiga pakkuda arendajale paindlik keskkond rakendusteegil *OpenGL* baseeruvate riistvaraliselt kiirendatud graafikasüsteemide prototüüpimiseks. Realiseeritud on järgnevad teenused:

- aknasüsteemi initsialiseerimine ja *OpenGL* renderdamiskonteksti loomine,
- tekstipõhisel identifikaatoritel baseeruv globaalne ressursside haldus:
    - muutujad:
        - andmetüübi määrab sisu,
        - lubatud andmetüübid: tekst, täisarv, murdarv;
    - joonistusprogrammid:
        - laadimine tekstifailist,
        - mallide kasutamise võimalus,
        - abstraktne liides programmide kasutamiseks,
        - toetatud programmide tüübid:
            - *GLSL* keelsed punkti-, geomeetria- ja fragmendiprogrammid,
            - konteinerprogrammid *GLSL* joonistusprogrammide kombineerimiseks;
    - tekstuurid;
- kahe- ja kolmemõõtmeliste piltide lugemine kõvakettalt ja tekstuuride loomine;
- pildi salvestamine (*TGA* formaadis), akna sisu salvestamine;
- paindlik sisendi töötlus – klaviatuuri ja hiire signaalide haldamine;
- kasutajaliidese elemendid:
    - kahemõõtmeliste rasterandmete töötluse dialoog,
    - kolmemõõtmelisest rasterandmete hulgast ühe lõigu valimise dialoog,
    - slaidide kuvamine,
    - statistika kuvamine (keskmine kuvatud raamide arv sekundis),
    - värvi valimise dialoog;
- konsool – liides, mis hõlmab järgnevat:
    - globaalsete ja rakendusespetsiifiliste käskude ühtne haldus,
    - kasutajaliides käsurea ning eelnevate käskude logi kuvaga võimaldades seeläbi lõppkasutajale:
        - ligipääsu globaalsetele muutujatele,
        - konsoolikäskude sisestamist;
- konsoolikäskude ajaliselt sünkroniseeritud taasesitamine,

- mitmed korduvalt kasutatavate funktsioonide realisatsioonid.

Raamistik kasutab mitmeid rakendusteeke ja baasklasse:

- *Cg Toolkit 2.0* – vabalt kasutatav kõrgtasemekeel graafikaprotsessori programmeerimiseks. Versioon 2.0 oli arenduse ajal kättesaadav vaid *Nvidia OpenGL SDK 10* installeerimispaketist [25];
- *Lua* – vabavaraline skriptimiskeel koos interpretaatoriga [19], [23];
- *GLEW* (*OpenGL Extension Wrangler Library*) – rakendusteegi *OpenGL* lisavõimaluste kasutamist lihtsustav avatud lähtekoodiga projekt [20];
- *GLUT* (*OpenGL Utility Toolkit*) – *Mark Kilgardi* operatsioonisüsteemist sõltumatu teek lihtsa *OpenGL* toega akendega programmi kirjutamiseks [21];
- *Nemo Framework* – *Cyril Crassini* loodud klassid tehete tegemiseks vektorite ja maatriksitega [7],
- *Simple Win32 Thread Class* – vabavaraline klass, mis kapseldab lõimede kasutamise operatsioonisüsteemis *WindowsXP* [27],
- *Texture Loader* – kahemõõtmeliste piltide laadimine püsimälu seadmetelt ja ettevalmistamine kasutamiseks rakendusteegis *OpenGL* (tekstuuride loomine) [22]. Teiste hulgas toetatakse järgnevaid sisendformaate:
  - *BMP – Windows Bitmap*,
  - *JPEG – Joint Photographic Experts Group*,
  - *PNG – Portable Network Graphics*,
  - *TGA – TrueVision TARGA*,
  - *GIF – Graphics Interchange Format*.

2.1. Joonistusprogrammide haldus

Järnevalt on ülevaade joonistusprogrammide kasutamisest globaalsete ressurssidena.

2.1.1. Joonistusprogrammid

Joonistusprogrammide laadimisel on tarvis teada, mis tüüpi programmiga on tegu. Programselt on võimalik tüüp määrata, kuid vaikimisi peab failinimes sisalduma viide:

- „_*VS*” – punktiprogramm,
- „_*GS*” – geomeetriaprogramm,
- „_*FS*” – fragmendiprogramm.

Geomeetriaprogrammide tekstis eeldatakse veel kolme lisaparameetri olemasolu, mis on vajalikud automaatseks joonistusprogrammide kombineerimiseks. Parameetri definitsioon on kujul „#nimi: väärtus”. Võimalikud nimed ning vastavad väärtused on :

- *GS_input* – defineerib geomeetriaprogrammi sisendi tüübi. Lubatud väärtused: *POINTS, LINES, LINES_ADJACENCY_EXT, TRIANGLES, TRIANGLES_ ADJACENCY_EXT*;
- *GS_output* – defineerib programmi väljundi tüübi. Lubatud väärtused: *POINTS, LINE_STRIP, TRIANGLE_STRIP*;
- *GS_count* – positiivne täisarv, millest rohkem punkte programm ühe sisendi korral ei väljasta. Maksimaalne väärtus oleneb kasutatavast riistvarast.

2.1.2. Konteinerprogammid

Konteinerprogrammid on üks globaalsete ressursside liik, mis kapseldavad joonistusprogrammide halduse koondades joonistusprogrammid komplektidesse. Gloobaalne ressursihaldus võimaldab taaskasutada samu joonistusprogramme eri konteinerprogrammides.

Konteinerprogrammi kirjelduse struktuuri elemendid on:

- „#” rea alguses tähistab kommentaari,
- viited joonistusprogrammi failidele:
  - rea alguses on laetava programmi tüüp, millele järgneb tühikuga eraldatud relatiivne otsingutee faili nimega
  - lubatud tüübid:
    - „*VS:*” – viide punktiprogrammile,
    - „*GS:*” – mittekohustuslik viide geomeetriaprogrammile,
    - „*FS:*” – viide fragmendiprogrammile.

```
#    Koch snowflake
#    input: POINT
VS: shaders/pass_through_VS.glsl
GS: shaders/snowflake_GS.glsl
FS: shaders/const_color_FS.glsl
```

Joonis 1. Näide konteinerprogrammist.

2.1.3. Mallide kasutamine

Joonistusprogrammide lugemisel töödeldakse failist loetavat teksti automaatselt: tuvastatakse metamärgendid, teostakse vastavad tegevused ja võimalusel salvestatakse väljund „*.prs*" laiendiga faili samas kaustas lähtefailiga. Märgend on ühe rea tekst, mis algab sümbolitega „*//#*". Toetatud on järgmised märgendite tüübid:

- *DEFINE* – malli deklaratsiooni algus. Parameetriks on loodava malli nimi. Igale *DEFINE* märgendile peab järgnema *END* märgend.
- *END* – malli deklaratsiooni lõpp.
- *INSERT* – Parameetris viidatud malli kasutamine.
- *INCLUDE* – Mallikogu lugemine kettalt.

```
Mallikogu test.shl:
     //#DEFINE test
     //#INSERT test
     c
     //#END
Lähtefail test.shd:
     a
     //#DEFINE test
     b
     //#END
     //#INCLUDE test.shl
     //#INSERT test
     d
Faili test.shd töötlemise väljund test.shd.prs:
     a
     b
     c
     d
```

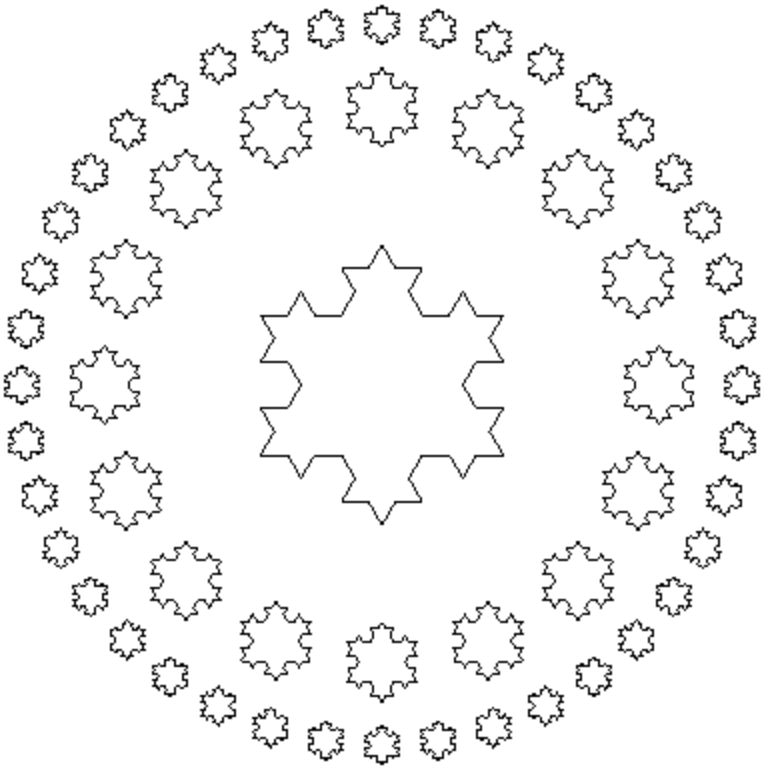
Joonis 2. Näide mallide kasutamisest.

2.2.Näited

OpenGL raamistiku kasutades on realiseeritud järgnevad programmid:

- *G80test* – kolmandas peatükis kirjeldatav Kochi lumehelbe joonistamine;
- *Tex3Dtrace* – neljandas peatükis kirjeldatud rastergraafika manipulaator;
- *CGFXtest* – lihtne näide graafikariistvara programmeerimiskeele *Cg* versiooni 2.0 uute võimaluste kasutamisest.

3. *Kochi* lumehelves



Joonis 3. Riistvaraliselt genereeritud *Kochi* lumehelbed

Kochi lumehelves [13] on fraktal, mille iteratiivset ehitamist alustatakse võrdkülgsest kolmnurgast, mille iga külje keskmine kolmandik asendatakse uue võrdkülgse kolmnurgaga. Viimase generatsiooni graafikasüsteemidel geomeetriaprogrammid võimaldavad üht lumehelvest kujutava joone genereerida dünaamiliselt graafika alamsüsteemis vastavalt nõutud iteratsioonide arvule.

Joonisel 3 on väljund programmi tööst, mis joonistab ette antud iteratsioonide arvuga Kochi lumehelbeid. Sisendandmeteks on iga helbe keskme koordinaat ning globaalne iteratsioonide arv, mida on võimalik jooksvalt muuta kasutades klahve „w” ja „s”. Iteratsioonide arvu murdarvuliste väärtuste korral kuvatakse arvu täisosale ning sellele järgnevale täisarvule vastava iteratsioonitaseme lineaarselt interpoleeritud kujund. Sel kombel on võimalik jälgida, kuidas kolmnurgast sammhaaval kujuneb lumehelves.

3.1.Algoritm

Kochi lumehelbe genereerimise algoritm:

- Punktiprogramm: edasta punkt muutmata kujul.
- Geomeetriaprogramm:
    - väljasta alguspunkt
    - kolmnurga iga külje jaoks:
        - kui iteratsioonide tase pole saavutatud, siis
            - jaota külg kolmeks,
            - keskmise lõigu asemele arvuta kaks uut
            - rakenda rekursiooni, neil kahel küljel
        - väljasta lõpppunkt
- Fragmendiprogramm: kirjuta välja interpoleeritud värv.

Esitatud algoritm, mille realisatsioon on toodud Lisas B, sobib hästi antud ülesannet lahendama, sest kõik lumehelbe tipud väljastatakse järjest, mis võimaldab need joonistada kasutades pidevat joont. See väldib korduvalt samade koordinaatide genereerimist, kui iga lõik eraldi joonistataks, mis annab ligi kahekordse kokkuhoiu mälu ribalaiuse kasutamisel.

3.2.Realisatsiooni kitsendused

Esimeseks takistuseks on asjaolu, et ka viimase generatsiooni riistvara programmeerimise keeled ei võimalda kasutada rekursiooni. Sellest saab mööda, kui käsitsi realiseerida rekursiivse alamprogrammi lokaalsete muutujate hoidmine vastavalt rekursiooni tasemele. Seda lähenemist rakendades on mõistlik kasutada kahemõõtmelist massiivi, mida indekseeritakse parasjagu töödeldava rekursiooni sügavuse ning muutuja indeksiga

Teise takistusena kerkib esile mitmemõõtmelisite massiivide toe puudumine. Seega tuleb kasutada veidi keerulisemat indekseerimist ühemõõtmelisse massiivi ja meeles pidada, kui kaugele ollakse igal tasemel jõutud.

Kolmas takistus on graafikprotsessori poolt maksimaalselt loodavate punktide arv ühe sisendi kohta, mis käesoleval juhul tähendab fraktali tippude arvu. *DirectX 10* standardi nõuete kohaselt on geomeetriaprogrammi väljund vähemalt 16 punkti. Täpse suuruse päringul väljastab *Nvidia 8800* seeria kaartitel *OpenGL* ohjur numbri 1024, kuigi riistvara toetab 128 punkti väljastamist. Kuna *Kochi* lumehelbe tippude arv on $3 * 2^{2*iteratsioonide\_arv}$, siis on võimalik teha kaks

iteratsiooni. Jätkamisel jääb üha suurem osa helvest moodustavast joone lõpust joonistamata.

3.3.Arendusprotsess

Arvestades eelmises punktis esitatud kitsendustega ja asjaoluga, et joonistusprogrammi pole võimalik otseselt siluda, kui neid graafikaprotsessoril käivitatakse, on mõistlik keerulisem arendus läbi viia standardseid programmeerimisvahendeid kasutades ning selle vaheetapi tulemus transleerida joonistusprogrammide kirjeldamise keelde. Lumehelbe arendusel oli arendusprotsessi sammude eesmärgid järgnevad:

- Kirjutada lihtsaim mõeldav geomeetriaprogramm, veenduda raamistiku töös.
- Luua geomeetriaprogramm, mis teeb ühe Kochi lumehelbe iteratsiooni sammu. Teha mõned testid.
- Luua abiprogrammid, mis on kirjutatud mõnes kõrgtaseme keeles (näiteks *C++*) ning realiseerida lumehelbe aluskolmnurga ühe külje iteratiivne konstruktsioon rekursiooni ja mitmemõõtmelisi massiive kasutamata. Veenduda programmi töökindluses piirjuhtudel. Antud lähenemise eeliseks on võimalused kasutada:
    - tekstilist väljundit, mis võimaldab kergelt veenduda andmete korrektsuses,
    - sammhaaval käskude täitmist.
- Kohandada kõrgtaseme keeles kirjutatud programm joonistusprogrammi meetodiks. Testida.
- Lisada joonistusprogrammile lumehelbe aluseks oleva kolmnurga nurkade koordinaadid ning rakendada neil paarikaupa eelnevalt loodud meetodid.

Arenduse lõpptulemusena valminud geomeetriaprogramm on toodus Lisas B ning seda kasutava rakendus *g80test.exe* võib leida laserplaadilt (vaata Lisa A).

4. Kolmemõõtmelise rastergraafika manipulaator

4.1. Idee

Alates kaheksakümnendatest aastatest, mil jõudsid tavakasutajateni esimesed akende süsteemil põhinevad graafilise kasutajaliidesega operatsioonisüsteemid, on olnud iseenesest mõistetavalt rakendusprogrammide hulgas vahendid digitaalsete piltide töötlemiseks algelistest vahenditest (näiteks *Microsoft Paint*) kuni tööstuslike ülivõimsate pakettideni (näiteks *Adobe Photoshop*).

Tavamõttes mõistetakse digitaalse pildi all kahemõõtmelist maatriksit, mille elemendid esitavad põhivärvide erksust antud punktis, luues seeläbi kujutise. Tajutava kujutise kvaliteeti mõjutavad peamiselt järgmised komponendid:

- Punkti suurus ja pildi mõõtmed – pildi laius ja kõrgus on määravad pildipunktide ehk pildimaatriksi elementide arvu. Keerukamate kujutiste esitamiseks on tarvis suuremõõtmelisi pilte. Teine kvaliteeti mõjutav aspekt on punktide arv pindala ühiku kohta – mida suurem see on, seda vähem on märgata diskretiseerimisel tekkivaid teravaid üleminekuid kõrvuti asetsevate punktide vahel.
- Värvisügavus – mõõdetakse bittides ühe pildipunkti kohta. Minimaalselt 1 bitt: selle abil saab esitada näiteks musta ja valget. Harilikult kasutatakse kaheksabitilist täisarvu ühe värvikomponendi kohta, mis vastab ühele baidile hallitoonilise pildil ning kolmele baidile ehk 24 bitile täisvärvis (ingl. *True Color*) pildi puhul. Vajadusel lisatakse täisvärvis pildile veel neljaski värvikanal, mis kannab läbipaistvuse infot. Kõrgema kvaliteedi saavutamise vajadusest tingituna kasutatakse värvivektoris täisarvulisi elemente suurusega 10, 12 või 16 bitti ning eriti täpse värviinformatsiooni korral ujukomarve täpsusega 16 ja 32 bitti, mis võimaldab realistlikumalt jäljendada inimese silma võimet kohanduda väga eredate ning väga tumedate värvide tajumiseks: päiksepaistelisest päevast kuuvalge ööni. Seda nimetatakse *HDR*-iks (ingl. *high dynamic range*).
- Esitluse kvaliteet – digitaalse pildi vaatamiseks võib selle paberkandjale printida, mis tähendab formaadi muutmist, kuid kõige mugavam on vaadata digipilti arvuti monitorilt või projektsiooniseinalt. Olenevalt väljundmeediast võib see olla suuteline esitama pilti täies kvaliteedis – nii värvi kui punkti tiheduse mõttes – või tuleb pildiinfol rakendada mõnd kohandamisprotseduuri. Näiteks harilikult on paberile võimalik printida rohkem inimsilma poolt

tajutavaid värve kui näidata monitoril. Kui printimise puhul ei ole oluline, kui kaua võtab kujutise paberile kandmine aega, sest hilisemat vaatamist see ei mõjuta, siis ekraanile kuvamise puhul võib osutuda vajalikuks esitlussüsteemi kiire toimimine. Näiteks juhul, mil pildi lahutus on oluliselt suurem monitoriga võrreldes, tuleb süsteemil pilti kas automaatselt vähendada või näidata vaid üht osa, samas võimaldades sujuvat üleminekut teiste regioonide vaatlemisele.

Rastergraafika esitamisel hoitakse kogu pilti harilikult muutmälus, mis seab piirid pildi mõõtmetele ning värvisügavusele. Näiteks kaasaegse digitaalse fotoaparaadi pildi harilik lahutus on 6 megapikselit, mis vastab kolme kaheksabitilise värvikanalitega pildile mõõtmetega 2816x2112. Selle pildi andmete hulk on üle 17 megabaidi. 10 aastat tagasi, oli taolises mahus digifotode töötlemine harilikult küllaltki vaevarikas, sest riistvaral puudus nii arvutusjõudlus vajalike transformatsioonide kiireks rakendamiseks kui ka muutmälu maht mitme pildi samaaegseks töötlemiseks või muudatuste ajaloo salvestamiseks ühe pildigi korral. 15 aasta eest oli personaalarvutite võimsust arvestades 800x600 mõõtmetest suuremate piltide kasutamine ebapraktiline.

Tänapäeval on võimalik kulutada ühe pildi jaoks vabalt pool gigabaiti mälu, ilma et süsteemi piirideni viia. Sellisele andmehulgale vastab ligikaudu näiteks 32-bitine digipilt mõõtmetega 16000x11185. Sama mahu võtab 512 punkti laiune kolmemõõtmeline kuup, mille iga 32-bitiline element sisaldab kolme värvikanali ja läbipaistvuse infot. Seega on olemas riistvara võimalused antud lahutusega rastergraafika praktiliseks laiemaks kasutuselevõtuks. Käesolevas peatükis kirjeldatakse töö raames realiseeritud tarkvarasüsteemi prototüüpi, mis võimaldab luua, muuta ning reaalajas kuvada kolmemõõtmelist rasterpilti. Süsteemi kasutusjuhend on toodud Lisas D. Hea ülevaate programmi väljundist annavad Lisas E toodud pildid, millest mõnede loomiseks kasutatud punktiprogrammide lähtetekstidega saab tutvuda Lisas C. Kõigi joonistusprogrammide terviktekstid leiab tööga kaasas olevalt laserplaadilt kaustast „*shaders*” nagu kirjeldatud Lisas A.

4.2.Redigeerimine

Kolmemõõtmelise andmehulga redigeerimine on taandatud kahemõõtmeliste piltide töötlemisele rastergraafika lihtsaimate vahenditega: esmalt valitakse välja koordinaattelgedega rööpne ristkülikukujuline läbilõige andmetest, millel on võimalik punkthaaval värvi muuta. Iga kahemõõtmelisel pildil tehtav muutus on koheselt jälgitav taustas nähtaval kolmemõõtmelise andmehulga kujutisel. Viilu valimine on realiseeritud vaid ühe telje suunas, kuid sisuliselt see piiranguid ei sea,

sest sisendandmeid saab 90° kaupa pöörata ümber keskpunkti mööda *x*-, *y*- või *z*-telge. Pööramise algoritmi koostamisel on aluseks [9], mida on laiendatud kolmemõõtmelisele juhule. Kui pööramine on abiks enne viilu valimist, siis peale seda saab kahemõõtmelise pildi redaktori sisu muuta valides viilu lähinaabreid.

4.3. Visualiseerimine

Kiire jälgimise meetodi (ingl. *ray-tracing*) kasutamisel luuakse kujutis „tulistades" suurel hulgal kiiri kaamerast punktis *C* erinevates suundades. Kolmemõõtmelist andmehulga rasteriseerimisel kujutatakse see ruumis risttahukana, mille suurus ning positsioon on kaamera suhtes vabalt valitavad. Poollõigul algusega *C*, mis lõikab antud tahukat on selle tahuka pinnaga kaks lõikepunkt siis, kui *C* asub väljaspool tahukat ja üks lõikepunkt, kui *C* asub tahuka sees. Olgu tahukasse siseneva või sealt algava poollõigu algus punkt *A*. Väljuv punkt olgu *B*.

4.3.1. Poolläbipaistvus

Poolläbipaistva kujutise konstrueerimiseks kolmemõõtmelisest andmehulgast kasutatakse kiire jälgimise algoritmi modifikatsiooni, kus summeeritakse proovid, mis võetakse piki kiirt kindla sammuga liikudes, ning jagatakse proovide arvuga ehk arvutatakse keskmine.

Sammude arv *S*, mis võetakse pildi konstrueerimisel ühe kiire jälgimisel läbi andmehulga sõltub kahest parameetrist: vabalt valitavast baasist S_c ning andmehulka siseneva ja sellest väljuva kiire punkti vahelisest kaugusest. Baasist sõltub otseselt tehtavate sammude arv. Kasutajal on võimalus seda süsteemi töö ajal jooksvalt muuta. Valides liialt väikse baasi, võib andmehulgast järjestikku võetavate proovide vahel olla liialt suured erinevused ja osa infost läheb kaotsi. Teise parameetri mõte on muuta sammude arv sõltuvaks tahuka orientatsioonist kaamera suhtes. Sammude arv on defineeritud järgnevalt:

$$S = S_c * \left|\overrightarrow{AB}\right|.$$

Poolläbipaistva kujutise loomiseks arvutatakse kiire poolt läbitud andmehulga keskmine väärtus võttes ühtlaselt *S* proovi andmehulgast *T*:

$$V = \frac{1}{S+1}\sum_{i=0}^{S} T\left(A + \frac{i}{S}(B-A)\right)$$

4.3.2. Pinna kujutamine

Loomaks kujutist pinnast antud andmehulgas, on tarvis liikuda piki lõiku *AB*, kuni leitakse esimene punkt *P*, mille optiline tihedus ρ_P on suurem ette antud piirväärtusest *c*:

$$P = \min_{A \to B}(\rho_P > c).$$

Käesoleva projekti korral on see tingimus kirjeldatud järgmiselt:

$$I(P) > 0.8,$$

kus funktsioon *I* leiab andmehulgast *T* argumendile vastava värvivektori komponentide summa:

$$I(P) = T(P).r + T(P).g + T(P).b.$$

Modifikaatorid *r*, *g* ja *b* vastavad leitud värvist punasele, rohelisele ja sinisele komponendile.

4.3.2.1. Puhas värv

Puhta värviga kujutamisel väljastatakse viimati andmehulgast võetud proovi väärtus:

$$V = T(P).$$

4.3.2.2. *Phongi* valgustusmudel

Antud mudeli aluseks on *Tuong Phongi* poolt kirjeldatud lokaalne empiiriline valgustusmudel, nagu kirjeldatud artiklites [14] ja [6]. Antud mudelit on laiendatud varjude lisamisega. Mudeli sisuks on pinna valgustatuse arvutamise eraldamine kolme komponendi summaks. Need komponendid on:

- taustvalgus (ingl. *ambient lighting*) – valgustatuse osa, mis ei sõltu valgusallikatest, vaid ainult pinna värvusest. See moodustab harilikult 10-20% kogu summast;
- difuusne valgus (ingl. *diffuse lighting*) – sõltub valgustatava pinna normaali ja valgustatava punkti ning valgusallika positsiooni vahelise vektori vahelise nurga koosiinusest, pinna ja valgusallika värvusest;
- spekulaarne valgus (ingl. *specular lighting*) – sõltub kaamera ning valgusallika suhtest pinna suhtes. Seda kasutatakse pinnale läike andmiseks.

4.3.2.2.1. Normaali arvutamine

Pinna normaal *N* punktis *P* on ühikvektor, mis arvutatakse diskreetse lähendiga funktsiooni *I* gradiendile. Selleks kasutatakse kolmemõõtmelist pilditöötlusfiltrit, mis võtab proovid punkti naabrusest ja arvutab funktsiooni *I* muutuste võrdlemisel normaali. Punkti *P* naabrid on defineeritud järgneva hulga suunavektoritega:

$$D = \{(1,0,0),(0,1,0),(0,0,1),(-1,0,0),(0,-1,0),(0,0,-1)\}.$$

Normaali arvutamiseks summeeritakse üle naabrite arvestades filtri parameetrit *r*, millega määratakse

$$N_s = \sum_{d \in D} d * \big(I(P + d * r) - I(P)\big)$$

Edasi jääb veel üle N_s ühikvektoriks teisendada:

$$N = \frac{N_s}{|N_s|}.$$

4.3.2.2.2. Varjudega arvestamine

Antud punkt *P* asub varjus valgusallika positsiooniga *L* suhtes, kui sirglõigul *PL* asub läbipaistmatu osa ehk eelnevalt kirjeldatud tingimust rahuldav punkt. Otsenähtavuse tuvastamiseks kasutatakse kiire jälgimise meetodit. Kuna punktis *P* on rahuldatud optilise piirtiheduse võrratus, siis on varju arvutamisel praktiliselt lähtuda punktist P_L:

$$P_L(P) = P + \frac{N}{(\overrightarrow{PL} \cdot N)} * r,$$

kus *N* pinna normaal punktis *P* ja *r* on sammu pikkus.

Punkti *B* nähtavus punktist *A* avaldub järgmiselt:

$$Vis(A,B) = \begin{cases} 1, & \overrightarrow{AP_L(A)} \cdot \overrightarrow{P_L(A)B} > 0 \wedge \max\limits_{p \in \overrightarrow{P_L(A)B}} I(p) < 0.8 \vee varjudeta \\ 0, & vastasel\ juhul \end{cases}$$

4.3.2.2.3. Lõpliku värvi arvutamine

Positsioonil L asuva valgusallika difuusne komponent on skalaar, mis avaldub järgmiselt:

$$V_{dif}(L) = \max\left(N \cdot \frac{\overrightarrow{PL}}{|\overrightarrow{PL}|}, 0\right) * T(P).$$

Positsioonil L asuva valgusallika spekulaarne komponent arvutatakse järgnevalt:

$$V_{spec}(L) = \max\left(\left(\frac{\overrightarrow{PC}}{|\overrightarrow{PC}|} \cdot R\left(\frac{\overrightarrow{PL}}{|\overrightarrow{PL}|}, N\right)\right)^{64}, 0\right),$$

kus R tähistab ühe vektori peegeldamist teise suhtes:

$$R(A,B) = A - 2B(A \cdot B).$$

Füüsikast on tuntud seaduspära, et valgusallika L poolt pinnaühiku kohta kiiratava valguse intensiivus väheneb võrdeliselt kauguse ruuduga. Antud efektiga arvestamiseks on defineeritud järgnev lähend, mille argumentideks on valgusallika ja valgustatava punkti positsioonid ning valgusallika mõju ulatus:

$$Dist(A,B,r) = \max\left(r - (B-A) \cdot (B-A), 0\right).$$

Eelneva alusel avaldub kujutatava pinna värvus V punktis P järgnevalt:

$$V = L_{amb} * T(P) + Vis(P,L)Dist(P,L,r)\left(L_{dif} * V_{dif}(L) + L_{spec} * V_{spec}(L)\right),$$

kus L_{amb} tähistab taustvalgustust, L_{dif} antud valgusallika poolt kiiratava valguse värvuse difuusset ja L_{spec} spekulaarse komponendi põhivärvide vektorit.

4.3.2.2.4. Mitu valgusallikat

Punkti P võib valgustada ka mitu valgusallikat. Antud juhul jääb taustvalgustus muutumatuks, kuid sellele lisanduvad erinevate valgusallikate difuussed ja spekulaarsed komponendid:

$$V = L_{amb} * T(P) + \sum_{L} Vis(P,L)Dist(P,L,r)\left(L_{dif} * V_{dif}(L) + L_{spec} * V_{spec}(L)\right)$$

4.3.3. Kiire jälgimise tehnikad

Kiire jälgimise meetod põhineb proovide võtmisel liikudes piki kiirt alates kaamerast kuni lõpmatuseni, korrates seda erinevaid suundi valides. Praktiliselt on olulised vaid proovid, mis pärinevad visualiseeritava andmehulga seest, seega on

esmane ülesanne leida punktid, mis tekkivad kiire lõikudes andmeid ümbritseva risttahukaga. Teine ülesanne on nende punktide vahel piki kiirt proovide võtmine.

4.3.3.1. Piiride arvutamine

Andmehulka ümbritseva risttahuka ja kiire lõiketest on realiseeritud kahel meetodil:

a) Kasutusel on lisapuhver, kuhu esimesel joonistuskorral salvestatakse kujutatavat andmehulka ümbritseva tahuka antud kaamera positsioonist lähtuvalt tagumistele külgedele vastavad tekstuuri koordinaadid. Teisel joonistuskorral kasutakse seda puhvrit ning joonistatakse kuubi eesmised küljed. Fragmendiprogrammis rekonstrueeritakse puhvri (kiire lõpppunkt) ja käesoleva sisendi (kiire alguspunkt) järgi kiire suund ning alguspunkt. Käesoleva tehnika aluseks on [2].
b) Kiire lõikepunktid arvutatakse fragmendiprogrammis kasutades *G. Scott Owen* poolt kirjeldatud tehnikat [X]. Kasutusel on *Nvidia SDK 10* [X] näite „*Render to 3D Texture*" põhjal kohandatud fragmendiprogrammi funktsioon, mis teostab kiire ning risttahuka lõiketesti kasutades maksimaalselt graafikaprotsessori vektortehteid. Antud tehnika eeliseks on asjaolu, et ei kasutata lisapuhvrit, mille tulemusel väheneb arvutusjõudluse arvelt mäluvajadus ning joonistuskordade arv. Samas pole kiire jälgimise seisukohalt praktiliselt oluline, kumba tehnikat kasutada, sest oluliselt suurem osa arvutusressursist kulub kiire jälgimisel proovide võtmiseks.

4.3.3.2. Kiire jälgimise kiirendamine

Kiire jälgimisel läbi kolmemõõtmelise andmehulga on kõige töömahukam osa paljude proovide võtmine. Esmane lähenemine on võtta proovid konstantse sammuga piki kiirt ühest lõikepunktist teise liikudes. Andmete piirkonnad, mis kujutuses esitavad läbipaistvust, ei mõjuta kiire jälgimise lõpptulemust. Eeldades, et need piirkonnad on piisavalt suured, on võimalik iga punkti jaoks sisend-andmetes välja arvutada minimaalne kaugus lähima olulise punktini. Seda suurust saab kasutada kiire jälgimisel sammu pikkusena, vältides asjatute proovide võtmist ebaolulisest regioonist. Tehnika aluseks on *William Donnely* artiklis [1] kirjeldatud kaugusfunktsioonide kasutamine: kasutatakse lisatekstuuri, millesse kodeeritakse lähima objekti kaugus antud ruumipunktis. Töömahukas kodeerimine viiakse läbi visualiseerimise eelsammuna, kujutise loomisel taaskasutatakse staatilist kauguste tekstuuri. Antud teadmine võimaldab fragmendiprogrammis liikuda keskmiselt

oluliselt pikema sammuga võrreldes konstantset sammu pikkus kasutades, mis viib kuni 35%-lise jõudluse kasvuni.

4.3.3.3. Varjude arvutamise kiirendamine

Sarnaselt eelmises punktis kirjeldatud kiire jälgimise kiirendamine on loomulik kasutada sama tehnikat varjude arvutamisel, mille käigus rakendatakse kiire jälgimise algoritmi valgustatava punkti ning valgusallika vahelisel lõigul. Arvutuse käigus kasutatav sammu pikkus proovide võtmisel sõltub pöördvõrdeliselt nende punktide vahelisest kaugusest ning igal sammul rakendatakse kaugusfunktsiooni sammu pikendamiseks. Antud tehnika võimaldab reaalajas arvutada kõrge kvaliteediga varjustatust, kuid see on käesolevat arvutusjõudlust arvestades veel siiski ebapraktiline.

Kiire lähend varjustatuse arvutamisel on konstantse arvu sammude tegemine olenemata valgusallika ning valgustatava punkti vahelisest kaugusest. Seda tehnikat kasutades ilmnevad vead varjude loomisel (20 sammu, 50 sammu) kuna tehakse liiga pikku samme, kuid samas luuakse võimalus lõppkasutajal kontrollida pildi kvaliteedi ja arvutustele kulutatava aja suhet. Näiteks valgusallika positsiooni animeerides liiguvad varjude piirid kiiresti, mis vähendab vaatleja poolt tajutavat müra.

4.3.4. Stereo

Kolmemõõtmelisest andmehulgast loodav kujutis on kahemõõtmeline pilt, kuid lähtuvalt kujutise loomise viisist, on seda kerge modifitseerida anaglüüf stereo (ingl. *anaglyph stereo*) kujutise genereerimiseks, millest läbi vastavate prillide vaadates on inimese aju võimeline rekonstrueerima pildi sügavusmõõtme. Antud efekt saavutatakse silmadele antava visuaalse informatsiooni eristamises: kohakuti on kaks pilti, mis on genereeritud erinevatest vaatepunktidest. Puna-rohelise anaglüüf stereo korral kasutatakse info eristamiseks värvikomponente kodeerides punasega vasaku silma poolt nähtava kujutise ning rohelise-sinisega parema silma tajutava. Pildi vaatamiseks tuleb kasutada värviliste klaasidega prille, mis blokeerivad teisele silmale mõeldud värvid. Anaglüüf stereo eelised võrreldes teiste sügavustaju tekitamise tehnikatega:

- keskmiselt hea kvaliteet,
- madal hind – ei nõua spetsiaalseid seadmeid,
- kergelt teisaldatav (vaid vaataja prillid),
- mitme kasutaja poolt vaadeldavad kujutised (igaühel omad prillid),
- pildi tiheduse täielik ärakasutamine (puudub üle rea joonistamine),

- silmadele keskmiselt koormav,
- kujutist saab trükkida paberile ja levitada.

Puna-rohelise anaglüüf stereo puudused on:

- prilliklaaside värvilised läätsed muudavad kujutise värve,
- puna-rohelise värvipimeduse korral sügavuse taju ei teki.

Sügavusinformatsiooniga pildi loomisel kasutatakse kahe joonistuskorraga algoritmi, mis tähendab topelt tööd ja poole väiksemat jõudlust:

- renderdada andmehulk lähtuvalt kaamera positsioonist *C*, väljastada vaid värvi punane komponent;
- teisel joonistuskorral nihutada kaamerat optilise peateljega paralleelselt paremale silmadevahelise kauguse võrra, eelmise joonistuskorra värvidele lisada käesoleva joonistuskorra roheline ja sinine komponent.

4.4. Jõudlus

Lähtuvalt süsteemi olemusest, on kasutaja jaoks oluline andmehulgas tehtavate muudatuste kiire kuvamine ning üldine interaktiivsus, vahetu suhtlemine süsteemiga, kasutatakse jõudluse hindamisel kasutatud sekundis genereeritavate kaadrite arvu (ingl. *frames per second*, *FPS*), millest sõltub ka sisendi töötlemine. Kuna jõudlus sõltub andmehulga mõõtmetest, loodava kujutise mõõtmetest, kaamera positsioonist ja vaatenurgast, valguse asukohast nii kaamera kui andmete suhtes, siis on mõõdetud jõudlus vaid hinnanguline.

Testimisel kasutatud riist- ja tarkvarakonfiguratsioon oli järgmine:

- 640MB mäluga *Nvidia 8800GTS* graafikakaart,
- graafikaohjuri versioon 97.92,
- 2GHz *AMD 64* keskprotsessor,
- 2GB muutmälu,
- Operatsioonisüsteem *WindowsXP*, lisaga *Service Pack 2*,
- renderdati pilte suurusega 1024x768 punkti.

Kujutamismeetod	***coords3d_%i.bmp***	***cornell_box.raw***
Poolläbipaistev	43	43
Puhas värv	42	36
Varjudeta *Phong*-valgustus	22	20
kaugusfunktsiooniga	46	52
Varjudega, 20 sammu	18	15
kaugusfuntsiooniga	36	30
kaks valgusallikat	27	18
Varjudega, 50 sammu	13	12
kaugusfuntsiooniga	26	21
kaks valgusallikat	16	10
Korrektsed varjud	11	10
Peegeldused	5	4

Tabel 1. Hinnanguline tööjõudlus (kaadrite arv sekundis)

Tebelis 1 esitatud suurusi võib kasutajakogemuse kvaliteedi järgi klassifitseerida järgnevalt:

- *FPS* väärtus on alla 15 – halb, süsteemi on raske kontrollida;
- 15 kuni 24 – keskpärane, süsteem on üldiselt kasutatav, kuid nõuab kasutajapoolset pingutust;
- enam kui 25 – hea, programm töötab sujuvalt, reageerides kasutaja sisendile viivitamatult, seda on mugav kasutada.

Lähtuvalt kasutusmugavusest saab Tabelis 1 toodu põhjal järeldada, et testisüsteem võimaldab kolmemõõtmeliste andmehulkade manipuleerimist ja visualiseerimist reaalajas kitsendustega kujutise loomise tehnika keerukusele. Arvestades graafikariistvara senist arengutempot, on poole aasta pärast loodud riistvara, mille kasutamine suurendab kõigi kasutatavate tehnikate kvaliteediklasse võrreldes käesoleva tasemega.

Kokkuvõte

Graafikariistvara uus generatsioon toob taas kord kaasa mitmeid funktsionaalseid uuendusi ning suurenenud jõudluse, mis loob võimalused üha arvutusmahukamate ja keerukamate visualiseerimisülesannete lahendamiseks reaal-ajas.

Unifitseeritud joonistusprogrammide arvutusüksuste kasutuselevõtuga on kõrvaldatud kitsaskoht, mis takistas mitmete algoritmide efektiivset realiseerimist, sest korraga ei olnud võimalik kasutada kogu arvutusjõudlust. Neljanda põlvkonna programmeeritavad graafikaprotsessorid võimaldavad vastavalt vajadusele balansseerida arvutusressurssi, mida näitlikustavad geomeetriaprogrammi kasutav Kochi lumehelveste genereerimine ning fragmendiprogrammis kiire jälgimise algoritmi kasutav rastergraafika manipulaator. Esimene rakenduse raskuskese on punktide, teise pikselite töötlemisel, kuid mõlemal juhul kasutatakse ära sama riistvara parandatud dünaamilise hargnemise realisatsiooni.

Autori panuseks on nende ainulaadsete ning uute näidete varal tutvustada ning avardada üldisi arusaamu graafikariistvarast, pakkudes innovatiivseid lahendusid tuntud ülesannetele, mida seni pole olnud võimalik realiseerida laiatarbe graafikariistvara kasutades, ning võimaldades seeläbi lauaarvutil graafikatööjaamadele omast funktsionaalsust.

Midagi nii lihtsat kui vaba käega joonistamine võib lähemal uurimisel kujutada arvutuslikult keerukat ülesannet, et säilitada mulje loomulikust ning enesestmõistetavast tegevusest. Rastergraafika manipulaator laiendab harilikku piltide loomist kolmandasse dimensiooni pakkudes võimalust tutvuda uue väljendusvõimalusega erinedes tavapärastest polügoonidel baseeruvatest modelleerimis-vahenditest. Lihtsa sisendliidese taustal on graafikariistvara võimsuse ning võimaluste piire kompav visualiseerimissüsteem, mis annab pidevalt tagasisidet pildis tehtavatest muudatustest kasutades erinevaid kujutustehnikaid. Vastavalt vajadusele saab valida lihtsast poolläbipaistvusest ja pindade värvi kuvamisest kuni varjude ning peegeldustega laiendatud *Phongi* valgustusmudelini. Suuremahuline sisend ning keerukamad algoritmid on siiski veel liialt nõudlikud, mille all kannatab kasutusmugavus, kuid peagi on oodata uusi ning veelgi võimsamat riistvara.

Graafikaprotsessorid muutuvad üha enam programmeeritavateks, laieneb nende kasutusotstarve. Üldisi arengutendentse jälgides on tulevik paralleelarvutuse päralt, milles graafikasüsteemid on harilike keskprotsessoritega võrreldes olulises eelisseisus, kuid on muutuste aeg. Head ideed leiavad kiiresti väljundi.

3D raster graphics on mainstream hardware
Abstract

Janek Press

This master's thesis research project takes in-depth view on 4th generation programmable mainstream graphics hardware such as Nvidia GeForce 8800 series as these implement Shader Model 4 which is part of Vista – latest operating system by Microsoft containing DirectX 10. Author's contribution is to demonstrate the new capabilities and introduce two novel approaches in real-time graphics on personal computers.

At first new functionality and capabilities of latest graphics hardware are described briefly followed by an overview of an OpenGL based framework to make use of recent advances. This easy to use framework developed by the author serves as a sandbox for prototyping two next generation graphics applications.

The first one shows off new geometry shader functionality by generating Koch snowflake fractal lines on the fly entirely on the GPU. CPU feeds just the center points of each snowflake to the graphics pipeline and the desired iteration level which can be changed dynamically – doing so, one can animate through different shapes. Implementing a recursive algorithm in an environment without support for neither recursion nor multidimensional arrays as provided by GLSL requires extra work.

Second proof-of-concept application implements author's vision of a basic editor with advanced visualization methods for editing three dimensional bitmaps. The system works in real-time enabling the user to immediately observe any changes. A range of various techniques are implemented get a presentation of the dataset: from transparency overview to Phong shading with addition of shadows and reflective surfaces. These shaders make heavy use of dynamic branching and require huge number of texture look-ups as GPU ray tracing is used to offload the CPU because generating high quality view of 3D data is far too complex for any PC. Performance of current hardware is sufficient in most common situations but further increase will enable more even complex lighting schemes.

Advances in 3D technology, both hardware and software, have brought workstation visualization capabilities to mainstream desktop computers. Finally it is possible to comfortably manipulate 3D raster graphics.

Allikad

Kasutatud kirjandus:

1. William Donnely. Per-Pixel Displacement mapping with Distance Functions. Artikkel raamatust: Matt Pharr, GPU Gems 2, leheküljed 123-136, 2005
2. J. Krüger, R. Westermann. Acceleration techniques for GPU-based volume rendering. IEEE Visualization 2003 materjalid, leheküljed 287–292; 2003
3. OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis. OpenGL Programming Guide: The Official Guide to Learning OpenGL, 2005
4. Janek Press. Bakalaureusetöö: Riistvaraline pilditöötlus, leheküljed 4-5, 13-14; 2004
5. Mark Tehver. Pildisünteesi meetodid (kursuse loengumaterjalid); 2006

Kasutatud kirjandus internetis (viited kehtivad seisuga 23. aprill 2007)

6. Michael V. Capps. Specular Reflection
   http://www.cs.nps.navy.mil/people/faculty/capps/4470/ads/specular.html
7. Cyril Crassin. OpenGL Geometry Shader Marching Cubes
   http://www.icare3d.org/content/view/50/9/
8. Hanners. NVIDIA G80 - Foxconn GeForce 8800 GTX review
   http://www.elitebastards.com/cms/index.php?option=com_content&task=view&id=228&Itemid=27
9. Michel Leunen. How to rotate a bitmap
   http://www.leunen.com/cbuilder/rotbmp.html
10. OpenGL Shading Language (GLSL) spetsifikatsioon
    http://www.opengl.org/documentation/glsl/
11. G. Scott Owen. Ray - Box Intersection
    http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm

12. Simon Stegmaier, Magnus Strengert, Thomas Klein. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting
http://www.vis.uni-stuttgart.de/ger/research/fields/current/spvolren/

13. Eric W. Weisstein. "Koch Snowflake." From MathWorld – A Wolfram Web Resource
http://mathworld.wolfram.com/KochSnowflake.html

14. Wikipedia. Phong reflection model
http://en.wikipedia.org/wiki/Phong_reflection_model

15. Wikipedia. Shader model comparison
http://en.wikipedia.org/wiki/HLSL

Internet (viited kehtivad seisuga 23. aprill 2007)

16. General-Purpose Computation Using Graphics Hardware
http://gpgpu.org/

17. ID Software. Quake III Arena Demo
ftp://ftp.idsoftware.com/idstuff/quake3/win32/Q3ADemo.exe

18. ID Software. Quake III Arena Source Code
ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip

19. Roberto Ierusalimschy, Waldemar Celes, Luiz Henrique de Figueiredo. Programmeerimiskeel Lua
http://www.lua.org/

20. Milan Ikits, Marcelo Magallon. OpenGL Extension Wrangler Library
http://glew.sourceforge.net/

21. Mark J. Kilgard. OpenGL Utility Toolkit
http://www.opengl.org/resources/libraries/glut/glut_downloads.php

22. Chris Leathley. Texture Loader
http://members.iinet.net.au/~cleathley/openGL/TextureLoader.htm

23. Lua Users Wiki. Programmeerimiskeel Lua - näited
http://lua-users.org/wiki/SampleCode

24. Microsoft. DirectX SDK - February 2007
http://www.microsoft.com/downloads/details.aspx?familyid=09f7578c-24aa-4e0a-bf91-5fec24c8c7bf&displaylang=en

25. Nvidia SDK 10, Nvidia SDK 9.5
http://developer.nvidia.com/object/sdk_home.html

26. OpenGL
http://www.opengl.org/

27. David Poon. Simple Win32 Thread Class
http://www.flipcode.com/cgi-bin/fcarticles.cgi?show=64032

Lisa A – *CD* sisukord

Tööle on lisatud *CD*, millel on:

- *CgFXtest\ – Cg 2.0* näiteprogrammi lähtekood,
- *Common\ – OpenGL* raamistiku lähtekood,
- *Data* – näiteprogrammide töökeskkond:
  - *Shaders* – joonistusprogrammid:
    - *Shaderlib.shl* – joonistusprogrammide mallide teek,
    - **.cg** – *Cg*-keelsed joonistusprogrammid,
    - **.glsl* – *GLSL*-keelsed joonistusprogrammid,
    - **.prg* – konteinerprogrammid,
    - **.prs* – malliparseri väljundfailid;
  - *Textures* – pildikujul kolmemõõtmelised andmehulgad, pildid;
  - *CgFXtest.exe – Cg 2.0* näiteprogramm,
  - *Cornell_box.raw* – kolmemõõtmeline andmehulk *Cornelli* kast,
  - *Cornell_name.raw* – kolmemõõtmeline andmehulk *Cornelli* kast tekstiga,
  - *Cube.lua* – ühest küljest avatud kuubi joonistamise *Lua* skript,
  - *Default.cfg* – *OpenGL* raamistiku üldine konfiguratsiooni kirjeldus,
  - *G80test.cfg* – *Kochi* lumehelbe renderdaja sätted,
  - *G80test.exe* – *Kochi* lumehelbe näiteprogramm,
  - *Pixel.demo* – kolmemõõtmelise rastergraafika joonistamise demo,
  - *Tex3dtrace.cfg* – kolmemõõtmelise rastergraafika manipulaatori sätted,
  - *Tex3dtrace.demo* – kolmemõõtmelise rastergraafika demo,
  - *Tex3dtrace.exe* – kolmemõõtmelise rastergraafika manipulaator;
- *G80test\ – Kochi* lumehelbe näiteprogrammi lähtekood,
- *Include\ – GLEW, GLUT* ja *Lua* teegipäised,
- *Lib\ – GLEW, GLUT* ja *Lua* teegifailid näiteprogrammide kompileerimiseks,
- *Tex3Dtrace* – kolmemõõtmelise rastermanipulaatori lähtekood,
- *Mastesr.doc* – käesolev dokument *Microsoft Office XP* formaadis,
- *Masters.pdf* – käesolev dokument *PDF* formaadis,
- *Masters.sln – Visual Studio 2005* tarkvaralahendus näiteprogrammidega,
- *TechDemo.avi* – rastermanipulaatori tehnoloogiate videotutvustus.

Lisa B – Kochi lumehelbe geomeetriaprogrammi lähtekood

```
/*
      Koch snowflake geometry shader by Janek Press
      POINT -> LINE STRIP
      parameters:
            fLevel - level of iterations
            fSize - size multiplier
      input: untransformed vertex position
      output: transformed vertexes, color white
      size of output: 3*2^(2*ceil(fLevel)) + 1

      #GS_input: POINTS
      #GS_output: LINE_STRIP
      #GS_count: 1024
*/

#version 120

#extension GL_EXT_geometry_shader4 : enable
#extension GL_EXT_gpu_shader4 : enable

uniform float fLevel;
uniform float fSize;

void Emit(vec2 a) {
      gl_FrontColor = vec4(1, 1, 1, 1);
      gl_Position = gl_ModelViewProjectionMatrix *
            gl_PositionIn[0].xyzw + vec4(a, 0, 0);
      EmitVertex();
}
```

```
void Link(int N, vec2 a, vec2 b, float levelMult) {
      if (N==0) {
        Emit(b);
        return;
      }

      vec2 stack[7*5];
      int levelProgress[7] = {4, 0, 0, 0, 0, 0, 0 };
      stack[3] = a;
      stack[4] = b;
      int level = 0;
      int level5 = 0;
      int count = 0;
      int max = 1 << (N*2);
      while (count<max) {
            if (level==N) {
                  for (int i=1;i<5;i++)
                        Emit(stack[level5+i]);
                  level--;
                  level5 -= 5;
                  count += 4;
            } else {
                  while (levelProgress[level]>4) {
                        level--;
                        level5 -= 5;
                  }
                  stack[level5+5  ] = stack[level5 + levelProgress[level]-1];
                  stack[level5+5+4] = stack[level5 + levelProgress[level]  ];
                  levelProgress[level]++;
                  level++;
                  level5 += 5;
                  levelProgress[level] = 1;
                  vec2 a = stack[level5];
                  vec2 b = stack[level5+4];
                  stack[level5+1] = mix(a, b, 0.33333);
                  float l = level==N ? levelMult : 1;
                  stack[level5+2] = mix(a, b, 0.5) +
                        l * 0.866025404/3*distance(b, a) *
                        cross( vec3(normalize(a-b),0), vec3(0, 0, 1) ).xy;
                  stack[level5+3] = mix(a, b, 0.66667);
            }
      }
}

void main(void) {
      int iLevel = int(fLevel+1);
      float levelMult = fLevel-floor(fLevel);

      vec2 triPoints[4] = {{0, 1}, {0.866, -0.5}, {-0.866, -0.5}, {0, 1}};
      Emit(triPoints[0]*fSize);
      for (int j=0;j<3;j++) {
            vec2 a = triPoints[j  ]*fSize;
            vec2 b = triPoints[j+1]*fSize;
            Link(iLevel, a, b, levelMult);
      }
      EndPrimitive();
}
```

Lisa C – Lähtekoodid kolmemõõtmelise rastergraafika peatükist

Teise joonistuskorra punktiprogramm

```
#version 120

uniform vec2 f2Size;
varying vec3 uvw;

void main(void) {
      vec4 v = ftransform();
      vec4 c = vec4(f2Size, 1, 1)*0.5;
      vec4 t = v + v.w;
      gl_Position = v;
      uvw = t.xyw*c.xyw;
      gl_FrontColor = gl_Vertex*0.5+0.5;
}
```

Poolläbipaistva kujutamise fragmendiprogramm

```
#version 120

#extension GL_ARB_texture_rectangle : enable

uniform sampler3D texData;
uniform int iSteps;
uniform vec4 backColor;
uniform vec3 cameraPos;

uniform sampler2DRect texBuf;
varying vec3 uvw;

void findOrigins(out vec3 Pfar, out vec3 Pnear, out vec3 Near, out vec3 Far)
{
      Pfar = texture2DRectProj(texBuf, uvw).xyz;
      Pnear = gl_Color.xyz;
      Near = Pnear*2-1;
      Far = Pfar*2-1;
}

void main(void) {
      vec3 Pfar, Pnear, Near, Far;
      findOrigins(Pfar, Pnear, Near, Far);
      float fMySteps = iSteps*length(Pfar-Pnear);
      vec3 P = Pnear;
      vec3 Pstep = (Pfar-Pnear) / fMySteps;
      vec4 col;
      int iSet = -1;
      for(int i=0; i<fMySteps; i++, P += Pstep) {
        col += vec4(texture3D(texData, P).rgb, 0);
      }
      gl_FragColor = col/fMySteps + backColor;
}
```

Puhta värviga kujutamise fragmendiprogramm

```
#version 120
#extension GL_ARB_texture_rectangle : enable

uniform sampler2DRect texBuf;
uniform sampler3D texData;
uniform int iSteps;
uniform vec4 backColor;
varying vec3 uvw;

void main(void) {
      vec3 B = texture2DRectProj(texBuf, uvw).xyz;
      vec3 A = gl_Color.xyz;
      float fMySteps = iSteps*length(B-A);
      vec3 s = (B-A)/fMySteps;
      vec4 col;
      for (int i=0;i<fMySteps;i++) {
            col += vec4(texture3D(texData, A).rgb, 0);
            if (length(col)>0.1) break;
            A += s;
      }
      gl_FragColor = col;
      if (length(col)<=0.1) gl_FragColor += backColor;
}
```

Phongi valgustusmudeli fragmendiprogramm

Järgnevas programmi lühikirjeldus:

- üks joonistuskord – kiire jälgimise algus ja lõpp arvutatakse jooksvalt,
- kaugusfunktsiooni kasutamine kiire jälgimisel,
- Phongi valgustusmudel,
- konstantse sammude arvuga leitud varjud (kiire, kuid defektidega),
- kaks valgusallikat.

```
#version 120

uniform sampler3D texData;
uniform int iSteps;
uniform vec4 backColor;
uniform vec3 cameraPos;
uniform vec3 lightPos;
uniform vec3 lightColor;
uniform vec2 shadowParams;
uniform vec3 sampleRange;
uniform vec3 textureSize;

varying vec3 RayOrign;
varying vec3 RayDir;

uniform sampler3D texDistance;
uniform float distanceScale;
```

```
float calcIntensity(sampler3D tex, vec3 coord) {
      return dot(texture3D(texData, coord).rgb, vec3(1, 1, 1));
}

vec3 sampleNormal(sampler3D tex, float base, vec3 coord, vec3 range, vec3
shift) {
      return shift * (calcIntensity(tex, coord+shift*range) - base);
}

vec3 findNormal(sampler3D t, vec3 point, vec3 range) {
      float base = calcIntensity(t, point);
      vec3 n = sampleNormal(t, base, point, range, vec3( 1,  0,  0));
      n +=   sampleNormal(t, base, point, range, vec3( 0,  1,  0));
      n +=   sampleNormal(t, base, point, range, vec3( 0,  0,  1));

      n +=   sampleNormal(t, base, point, range, vec3(-1,  0,  0));
      n +=   sampleNormal(t, base, point, range, vec3( 0, -1,  0));
      n +=   sampleNormal(t, base, point, range, vec3( 0,  0, -1));
      return normalize(n);
}

bool isVisible(sampler3D tex, vec3 A, vec3 B) {
      vec3 Pstep = (B-A)/shadowParams.g;
      vec3 P = A + Pstep;
      for (int i=0;i<shadowParams.g;i++,P+=Pstep)
            if (calcIntensity(tex, P)>0.8) return false;
      return true;
}

vec4 doPhong(sampler3D tex, vec4 sample, vec3 range, vec3 P, vec3 lightPos) {
      vec4 col = vec4(vec3(0.1, 0.1, 0.1)*sample.rgb, 1);
      vec3 vToLight = P - lightPos;
      float fAttenuation = 0.7 - dot(vToLight, vToLight);
      if (fAttenuation>0) {
            vec3 N = findNormal(tex, P, range);
            vec3 lightDir = normalize(vToLight);
            float fDiffuse = dot(lightDir, N);
            if (fDiffuse>0) {
                  bool bDoLighting = true;
                  if (0.9<shadowParams.r) {
                        vec3 Plight = P - lightDir/fDiffuse*range;
                        if (dot(lightPos-P, lightPos-Plight)>0)
                              bDoLighting = isVisible(tex, Plight, lightPos);
                  }
                  if (bDoLighting) {
                        col.rgb += fAttenuation*fDiffuse*
                              lightColor*sample.rgb;
                        float fSpecular = dot(normalize(cameraPos-P),
                              reflect(lightDir, N));
                        if (fSpecular>0) col.rgb +=
                              fAttenuation*pow(fSpecular, 64);
                  }
            }
      }
      return col;
}
```

```
bool IntersectBox(vec3 RayOrign, vec3 RayDir, vec3 boxmin, vec3 boxmax, out
float tnear, out float tfar) {
      // compute intersection of ray with all six bbox planes
      vec3 invR = 1.0 / RayDir;
      vec3 tbot = invR * (boxmin.xyz - RayOrign);
      vec3 ttop = invR * (boxmax.xyz - RayOrign);

      // re-order intersections to find smallest and largest on each axis
      vec3 tmin = min (ttop, tbot);
      vec3 tmax = max (ttop, tbot);

      // find the largest tmin and the smallest tmax
      vec2 t0 = max (tmin.xx, tmin.yz);
      float largest_tmin = max (t0.x, t0.y);
      t0 = min (tmax.xx, tmax.yz);
      float smallest_tmax = min (t0.x, t0.y);

      // check for hit
      bool hit;
      if ((largest_tmin > smallest_tmax))
            hit = false;
      else
            hit = true;

      tnear = largest_tmin;
      tfar = smallest_tmax;

      return hit;
}

void findOrigins(out vec3 Pfar, out vec3 Pnear, out vec3 Near, out vec3 Far)
{
      vec3 boxMin = vec3(-1, -1, -1);
      vec3 boxMax = vec3( 1,  1,  1);

      RayDir = normalize(RayDir);

      float tnear, tfar;
      bool hit = IntersectBox(RayOrign, RayDir, boxMin, boxMax, tnear, tfar);
      if (!hit) discard;
      if (tnear < 0.0) tnear = 0.0;

      // calculate intersection points
      Near = RayOrign + RayDir*tnear;
      Far = RayOrign + RayDir*tfar;
      // convert to texture space
      Pnear = Near*0.5 + 0.5;
      Pfar = Far*0.5 + 0.5;
}

uniform vec3 lightPos1;

void main(void) {
      vec3 Pfar, Pnear, Near, Far;
      findOrigins(Pfar, Pnear, Near, Far);
      float fMySteps = iSteps*length(Pfar-Pnear);
```

```
	vec3 P = Pnear;
	vec3 Pstep = (Pfar-Pnear) / fMySteps;
	vec4 col;
	int iSet = -1;
	float fSet = -1;
	for(float i=0; i<fMySteps; i++, P += Pstep) {
		vec4 sample = texture3D(texData, P);
		if (length(sample.rgb)>0.8) {
			fSet = i;
			col = doPhong(texData, sample, sampleRange, P, lightPos);
			col += doPhong(texData, sample, sampleRange, P, lightPos1)-
				vec4(0.1, 0.1, 0.1, 1);
			break;
		} else {
			float d = texture3D(texDistance, P).a *
				iSteps * distanceScale;
			i += d;
			P += d*Pstep;
		}
	}

	if (fSet<0) discard;
	gl_FragColor = col;
	vec4 T = gl_ModelViewProjectionMatrix*
		vec4(Near+(Far-Near)*fSet/fMySteps, 1);
	gl_FragDepth = T.z/T.w*0.5+0.5;
}
```

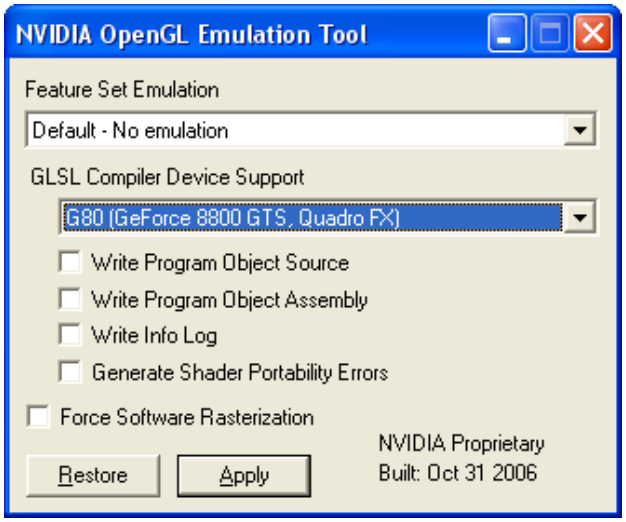
Korrektne nähtavuse funktsioon

Joonistusprogrammis *trace_shadow.prg* on kasutusel järgnev nähtavuse funktsioon, mis võtab kiire jälgimise sammu pikkuse arvutamisel arvesse punktide vahelist kaugust ning kiirendab sammude tegemist kaugusfunktsiooniga. Antud versioon on loogiliselt keerukam ja tavasituatsioonis töömahukas, kuid on vaba eelnevalt kasutatud kiire lähendi vigadest (korrektsed varjud, lähend 20 ja 50 sammuga).

```
bool isVisible(sampler3D tex, vec3 A, vec3 B) {
	vec3 a = A;
	float fSteps = iSteps*length(B-a);

	vec3 Pstep = (B-a)/fSteps;
	vec3 P = a;
	for (float i=0;i<fSteps;i++,P+=Pstep) {
		if (calcIntensity(tex, P)>0.8) return false;
		else {
			float d = texture3D(texDistance, P).a * iSteps;
			i += d;
			P += d*Pstep;
		}
	}
	return true;
}
```

Lisa D – Kolmemõõtmelise rastergraafika manipulaatori kasutusjuhend



Joonis 4. *Nvidia Emulation tool*

Enne programmi *tex3dtrace.exe* käivitamist, on oluline veenduda normaalseks tööks vajalikes tingimustes:

- kaasaegse graafikakaardi olemasolu (soovitavalt *Nvidia 8800*), vajalik on *OpenGL 2.1* täiemahuline riistvaraline tugi,
- kõigi vajalike ohjurite korrektne paigaldatus, vajalik on *GLSL* kompilaatori tugi keele versioonile 1.2,
    - *Nvidia* graafikakaartide korral on vajalik peale ohjurite (arenduses kasutati versiooni 97.92) paigaldust muuta *GLSL* kompilaatori profiili, et see toetaks 8800 seeria kaarte (vaikimise on pikad joonistusprogrammid välja lülitatud), milleks saab kasutada arendusvahendit *Nvidia OpenGL Emulation Tool* (Joonis 4).

Programmi käivitamisel töödeldakse esmalt konfiguratsioonifailid *default.cfg* ja *tex3dtrace.cfg*, kus on kirjeldatud globaalsete muutujate algväärtused ning seotakse kasutaja sisend programmsete tegevustega. Esimese parameetrina võimalik ette anda:

- kolmemõõtmelise pildi nimi, mida soovitakse avada. Vaikimise kasutatakse väärtust „*textures/coords3d_%i.bmp*”;
- konfiguratsioonifaili nimi, mis täidetakse peale alglaadimist;

Süsteemi töö ajal saab alati avada konsooli klahvi „~” vajutades. Konsoolis on näha logi ja käsurealt on võimalik sisestada käske.

Konsoolis kasutatavad klahvid:

- järgmine lehekülg – logi kerimine üles vanema väljundi vaatamiseks,
- eelmine lehekülg – logi kerimine alla uuema väljundi vaatamiseks,
- nool üles – eelnevalt sisestatud käsu valimine,
- nool alla – käskude ajaloos ette poole liikumine,
- tabulatsiooniklahv – osalise käsu põhjal sarnaste käskude kuvamine,
- sisestusklahv – sisestatud käsu täitmine,
- paoklahv – konsooli sulgemine, suundumine tagasi eelnevasse seisundisse.

Konsooli käsud:

- „*clear*" – konsooli logi kustutamine,
- „*load*" – kolmemõõtmelise pildi laadimine. Argumendid:
    - kataloogi ja failinimi ühe stringina.

      Kui failinimes sisaldub tähekombinatsioon „*%i*", siis asendatakse see numbritega alates nullist – antud juhul on kolmemõõtmeline andmehulk kirjeldatud harilike piltide seeriana, millel on lubatud kõik toetatud sisendformaadid. Vastasel juhul kasutatakse binaarset pakkimata formaati järgneva struktuuriga:

        - 2 baiti formaadi tähis: „*R3*",
        - 3*2 baiti andmehulga mõõtmed (laius, kõrgus, sügavus),
        - andmehulga sisu (*z*, *y* ja *x* mõõtmes) neljabaidiste värvikirjetena;
- „*lua*" – *LUA* skripti käivitamine kettalt. Argumendid:
    - skriptifaili nimi.

      Alternatiivne võimalus *LUA* skriptide käivitamiseks on „.*lua*" laiendiga failinime sisestamine konsoolikäsuna.

- „*new*" – 3D pildi loomine. Argumendid on täisarvulised:
    - laius,
    - kõrgus ja
    - sügavus;
- „*play*" – konsooli käskude taasesitamine ehk demo mängimine,
- „*save*" – kolmemõõtmelise pildi salvestamine. Argumendid:
    - kataloogi ja failinimi ühe stringina,
- „*quit*" – programmist väljumine.

Vaatlusseisund (vaikeolek programmi käivitudes):

- „*c*” – taustavärvi muutmise dialoogi kuvamine,
- „*f*” – võrekujundi joonistamise sisse-välja lülitamine,
- „*k*” – üldine kvaliteedi muutmine,
- „*l*” – varjude arvutamise lülitamine *Phongi* valgustusmudeli korral,
- „*m*” – sisendandmete silumise sisse-välja lülitamine,
- „*p*” – aktiivse joonistusprogrammi vahetamine,
- „*t*” – anaglüüf stereo vaate sisse-välja lülitamine,
- „*v*” – kuvab kolmemõõtmelisest pildist ühe lõigu valimise dialoogi,
- „*x*” – kahekordse suurenduse sisse-välja lülitamine,
- „*w*” ja „*s*” – esitatava kujundi kaugemale-lähemale toomine,
- „*z*” – valgusallika positsiooni juhtimine *Phongi* valgustusmudeli korral,
- funktsiooniklahvid 1 kuni 11 – joonistusprogrammi aktiveerimine:
    - 1 kuni 3 – kahe joonistuskorraga programmid:
        - 1 – poolläbipaistvus,
        - 2 – puhas värv,
        - 3 – laiendatud Phongi valgustusmudel varjudega;
    - 4 kuni 11 – ühe joonistuskorraga programmid:
        - 4 kuni 6 – harilikku sammumist kasutavad programmid:
            - 4 – poolläbipaistvus,
            - 5 – laiendatud *Phongi* valgustusmudel varjudega;
            - 6 – laiendatud *Phongi* valgustusmudel varjudega, lisatud poolläbipaistvus;
        - 7 kuni 11 – kaugusfunktsiooni kasutavad programmid:
            - 7 – *Phongi* valgustus varjudega,
            - 8 – korrektsed varjud,
            - 9 – kahekordne vaade: lineaarselt interpoleeritud sisendandmetele on poolläbipaistvalt lisatud lähima proovi filtreerimisel saadud kujutis,
            - 10 – kaks valgusallikat *Phongi* valgustuses,
            - 11 – peegeldused;
- numbriklahvid 5 kuni 7 – andmehulga keeramine 90° vastavalt ümber *x*-, *y*- või *z*-telje,
- numbriklahvid 8, 9, 0 – laetakse kolmemõõtmeline näidisandmehulk:
    - *textures/coords3D_%i.bmp*,
    - *cornell_box.raw*,
    - *cornell_name.raw*;
- paoklahv – programmi sulgemine,

- hiire liigutamine vasakut nuppu all hoides – kujutise keeramine.

Viilu valimise dialoog:

- paoklahv – suundumine tagasi vaatlusseisundisse,
- hiire liigutamine mõne viilu kohal – tõstab esile ühe viiludest,
- hiire vasaku nupu vajutus viilul – valitud viilu asutakse manipuleerima.

Viilu manipuleerimine:

- „*a*" ja „*d*" – joonistuse nihutamine joonistusalal horisontaalselt,
- „*c*" – aktiivse värvi muutmise dialoogi kuvamine,
- „*h*" ja „*j*" – joonistusala vähendamine-suurendamine,
- „*q*" ja „*e*" – aktiivse viilu muutmine,
- „*w*" ja „*s*" – joonistuse nihutamine joonistusalal vertikaalselt,
- numbriklahvid 1 kuni 4 – joonistuspea suuruse seadmine 1 kuni 4 punkti
- paoklahv – suundumine vaatlusseisundisse,
- hiire horisontaalne liigutamine paremat klahvi all hoides – joonistusala sisu suurenduse muutmine,

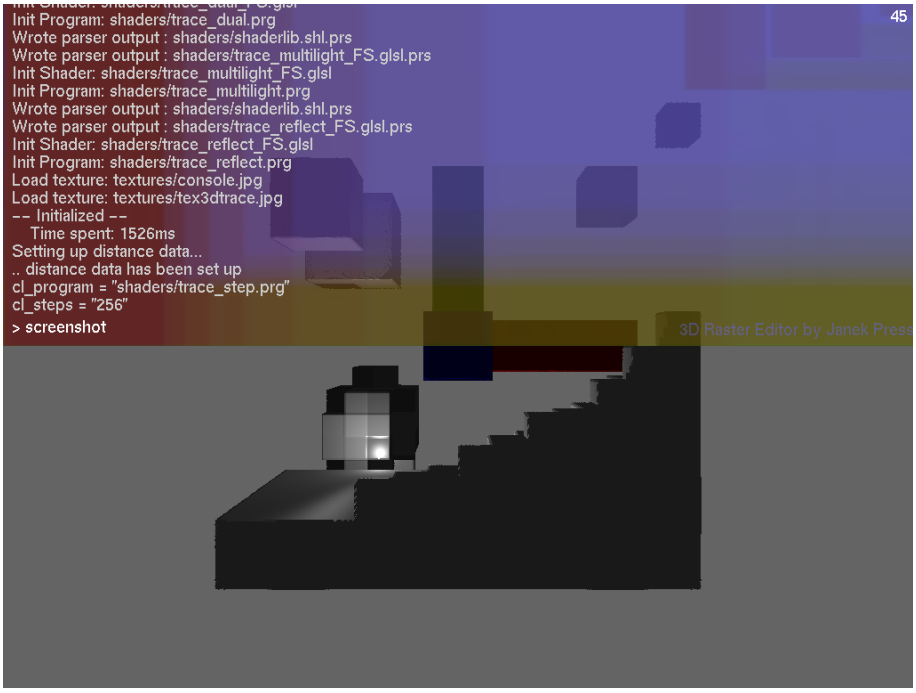
Värvi muutmise dialoog:

- paoklahv – suundumine tagasi eelnevasse seisundisse,
- hiire vasaku nupu vajutus:
    - värvitulbal – muudab vastava värvikomponendi eredust,
    - rohelisel nupul – kinnitab valiku, suundumine tagasi eelnevasse seisundisse,
    - punasel nupul – uue värvi valikust loobumine, suundumine tagasi eelnevasse seisundisse.

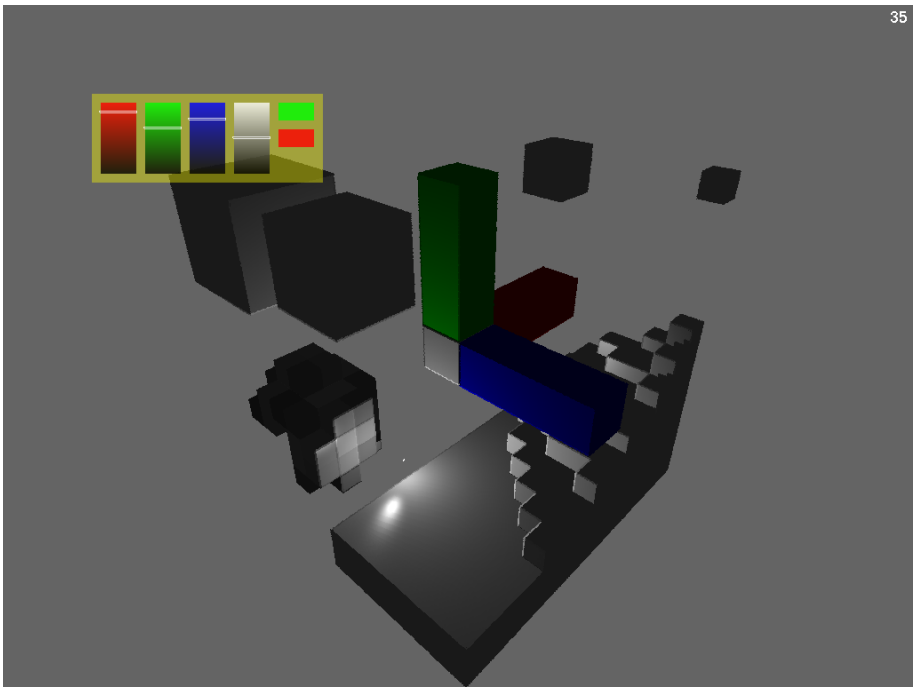
Mõned tähtsamad globaalsed muutujad, mille väärtust mõjutab pildi kvaliteeti:

- *cl_distanceScale* – kaugusfunktsiooni kordaja kiire jälgimisel,
- *cl_sampleRange* – proovide võtmise raadiuse kordaja normaali arvutamisel,
- *cl_shadowSteps* – sammude arv varju arvutamisel,
- *cl_steps* – minimaalne sammude arv kiire jälgimisel

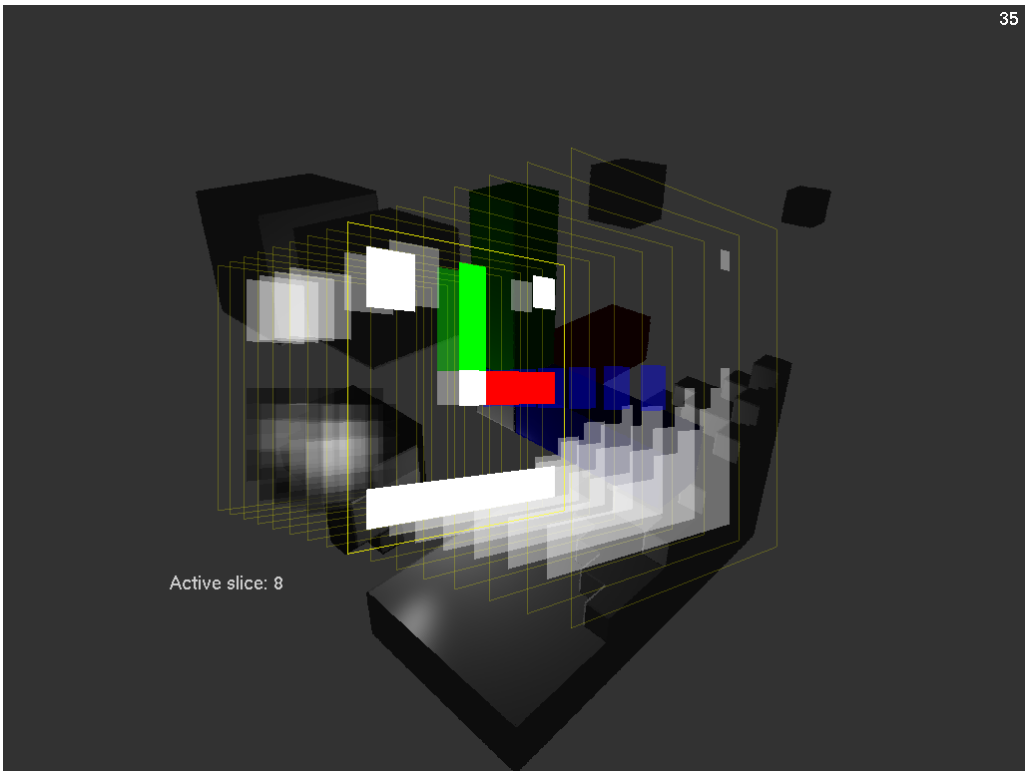
Lisa E – Kolmemõõtmelise rastergraafika manipulaatori vaated



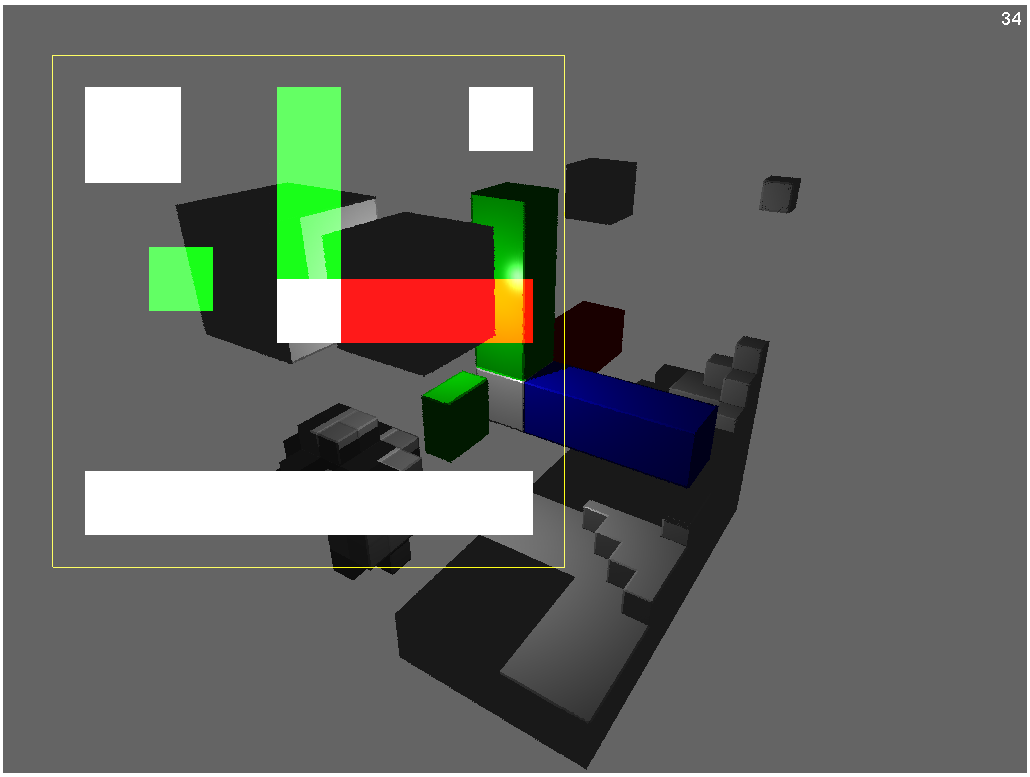
Konsool



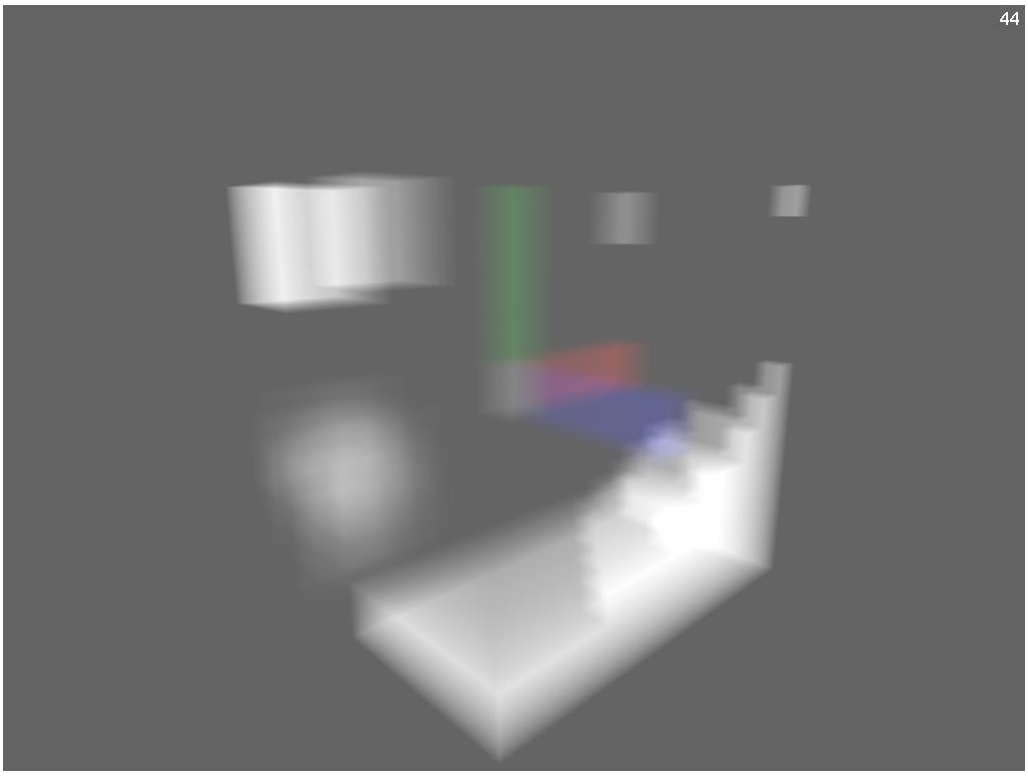
Värvi valimise dialoog



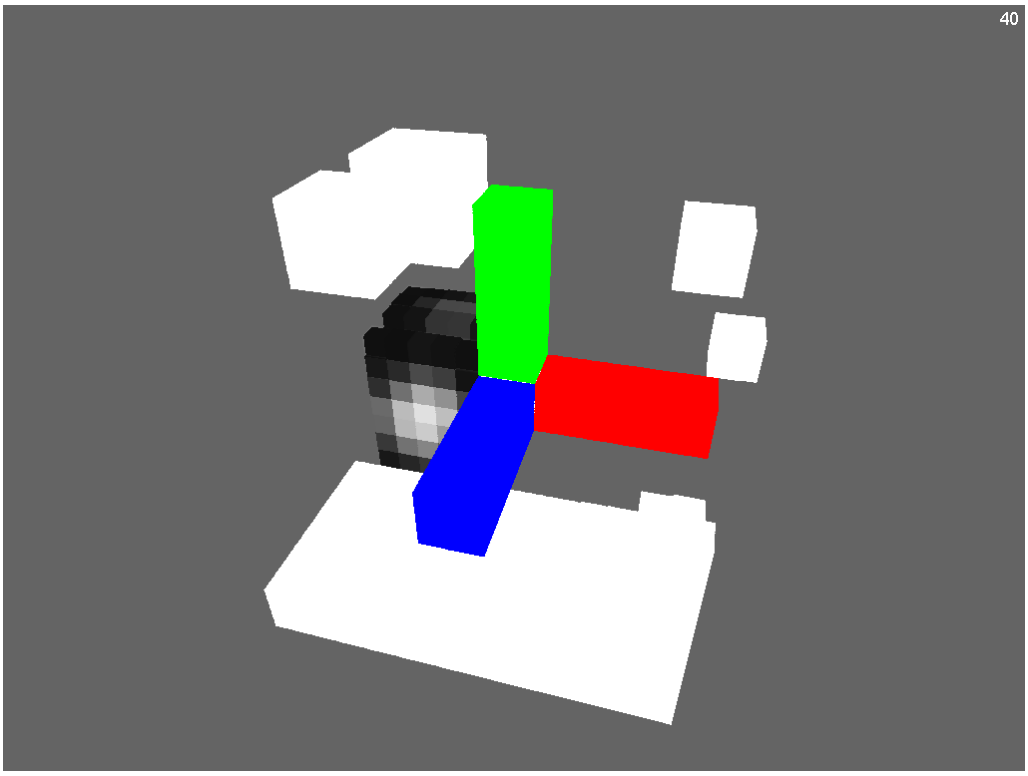
Viilu valimise dialoog



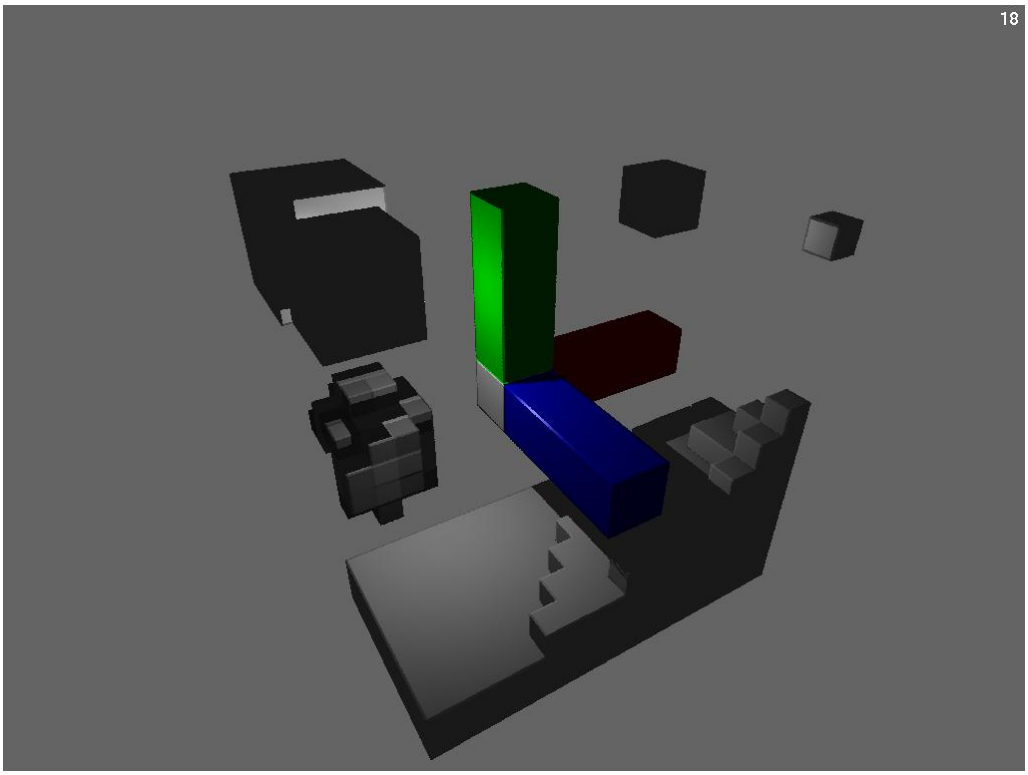
Viilu redigeerimine



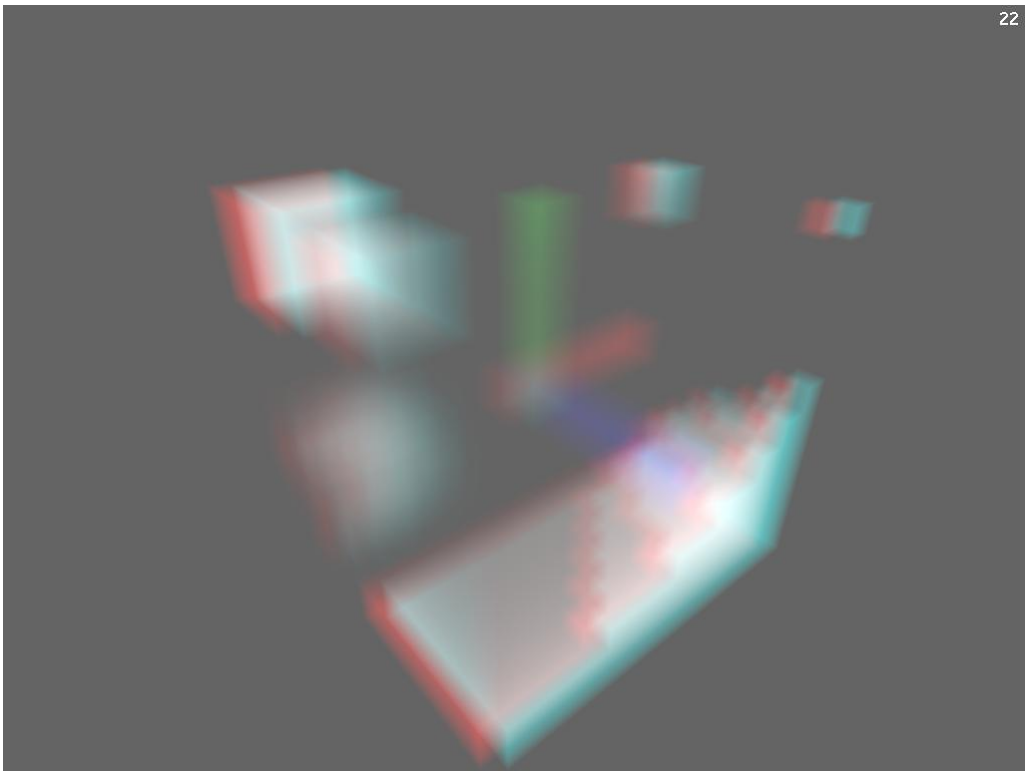
Poolläbipaistvad objektid (kaks joonistuskorda)



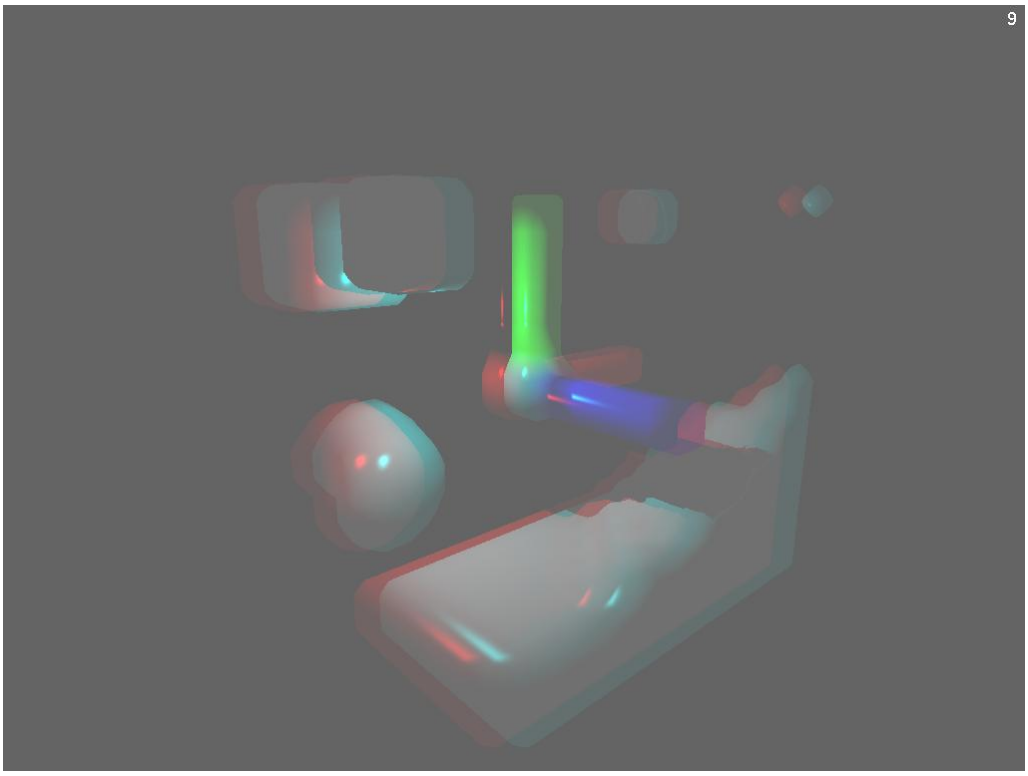
Puhta värviga objektid (kaks joonistuskorda)



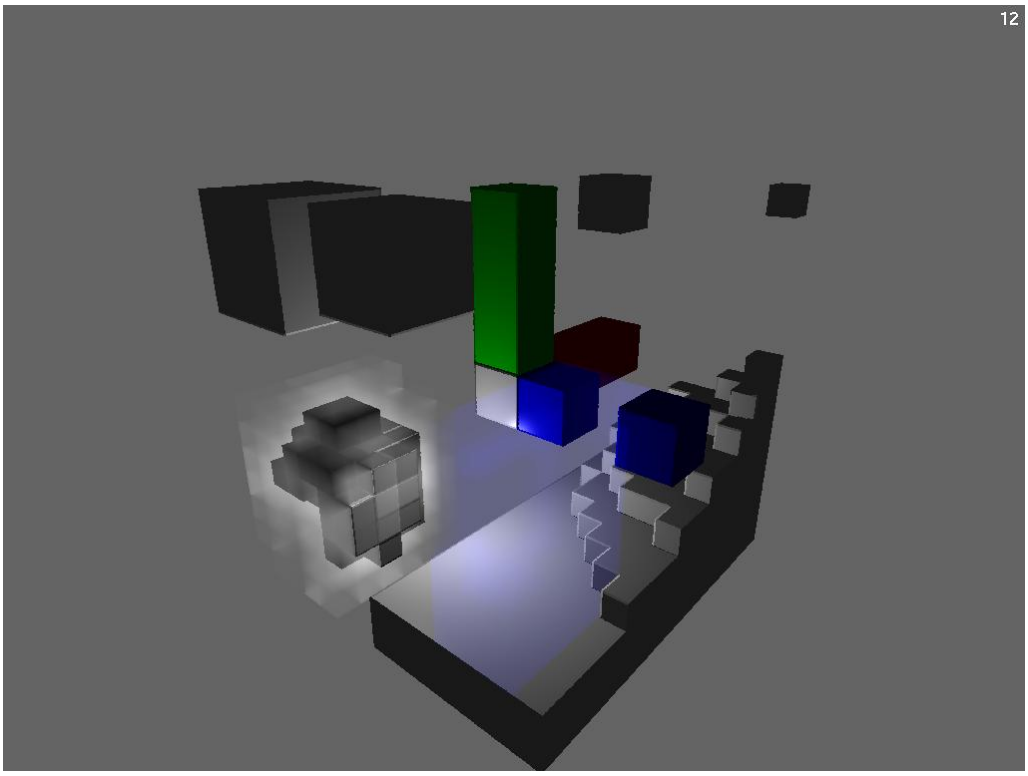
Laiendatud *Phongi* valgustusmudel (kaks joonistuskorda)



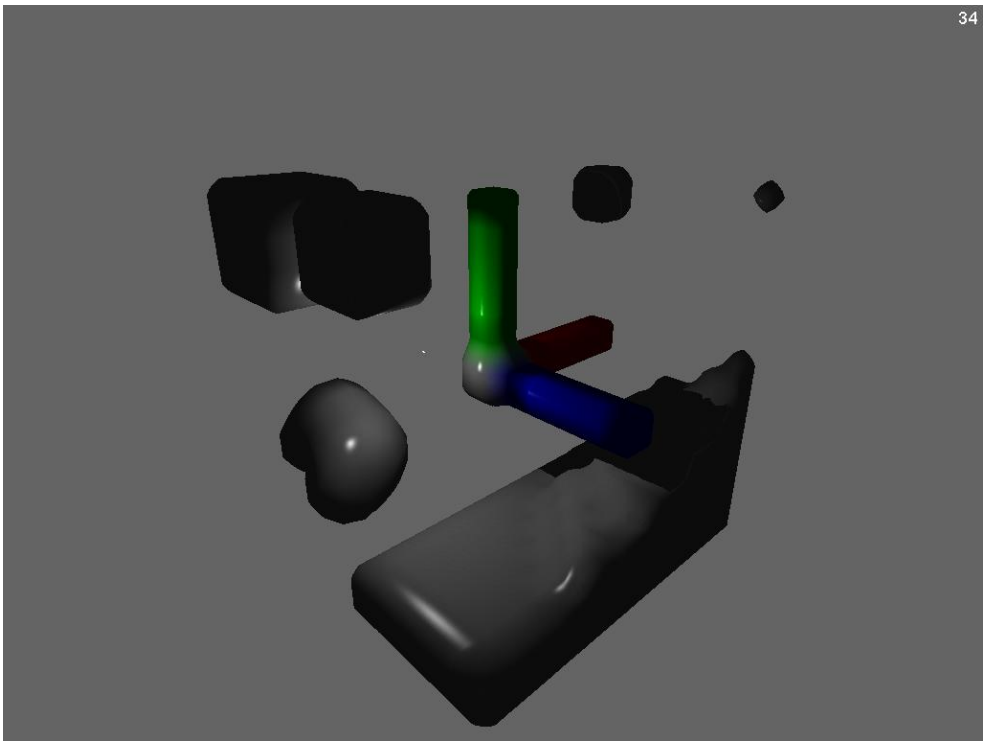
Poolläbipaistvad objektid (üks joonistuskord, stereo)



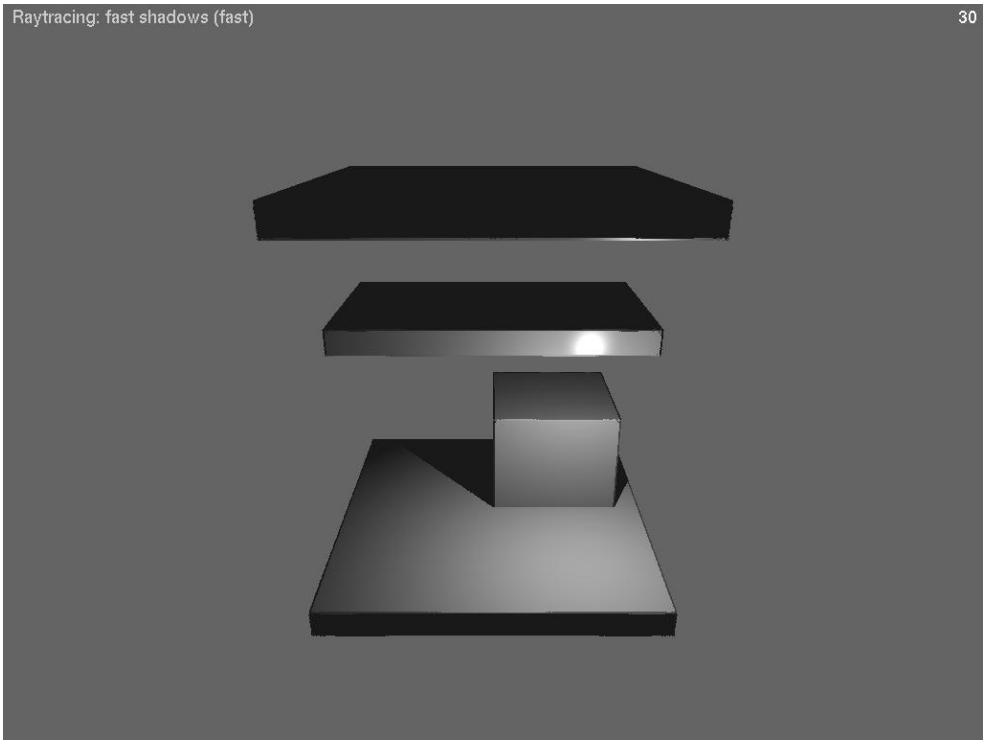
Laiendatud *Phongi* valgustusmudel (üks joonistuskord, stereo, sisendi silumine)



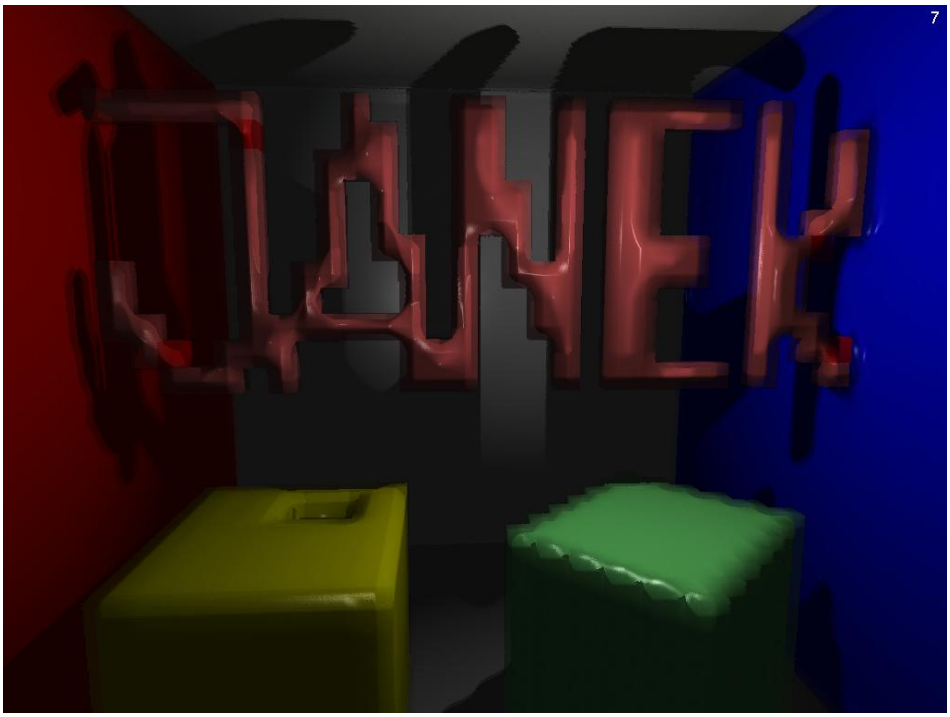
Laiendatud *Phongi* valgustusmudel poolläbipaistvusega (üks joonistuskord)



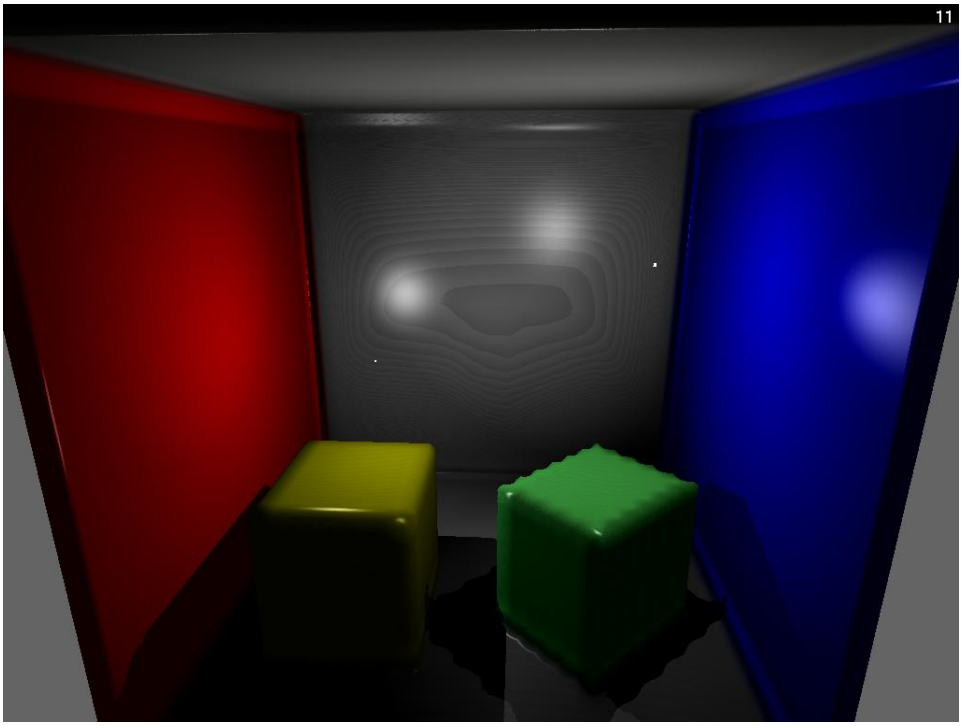
Kiirendatud valgustusmudel (kaugusfunktsiooniga kiirendatud, üks joonistuskord, sisendi silumine)



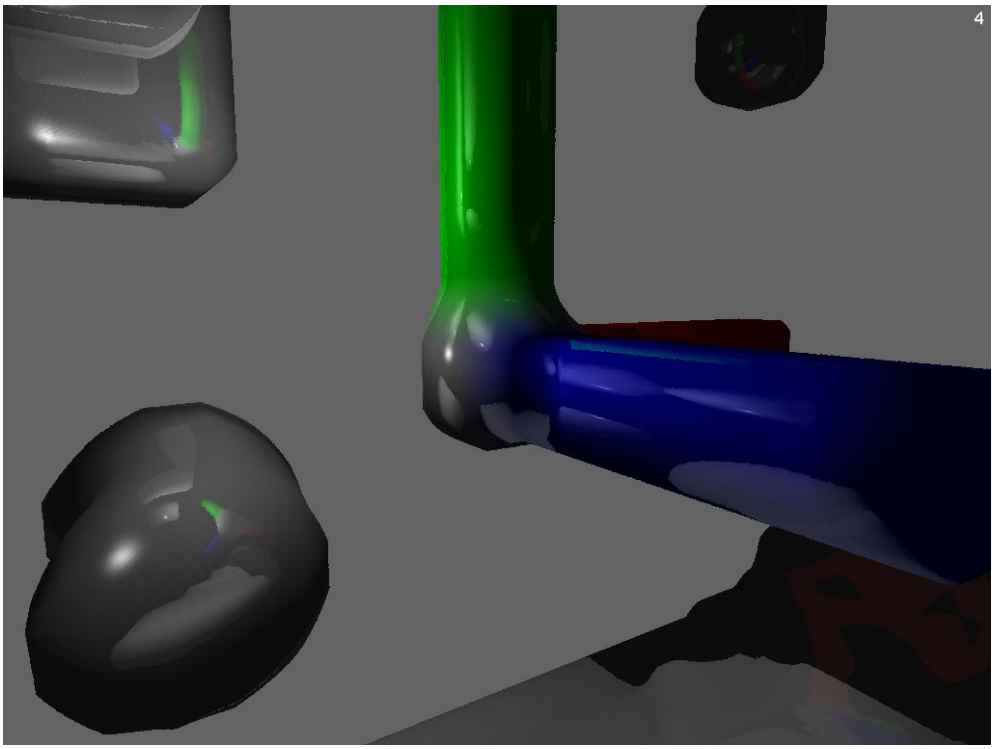
Korrektsed varjud (kaugusfunktsiooniga kiirendatud, üks joonistuskord)



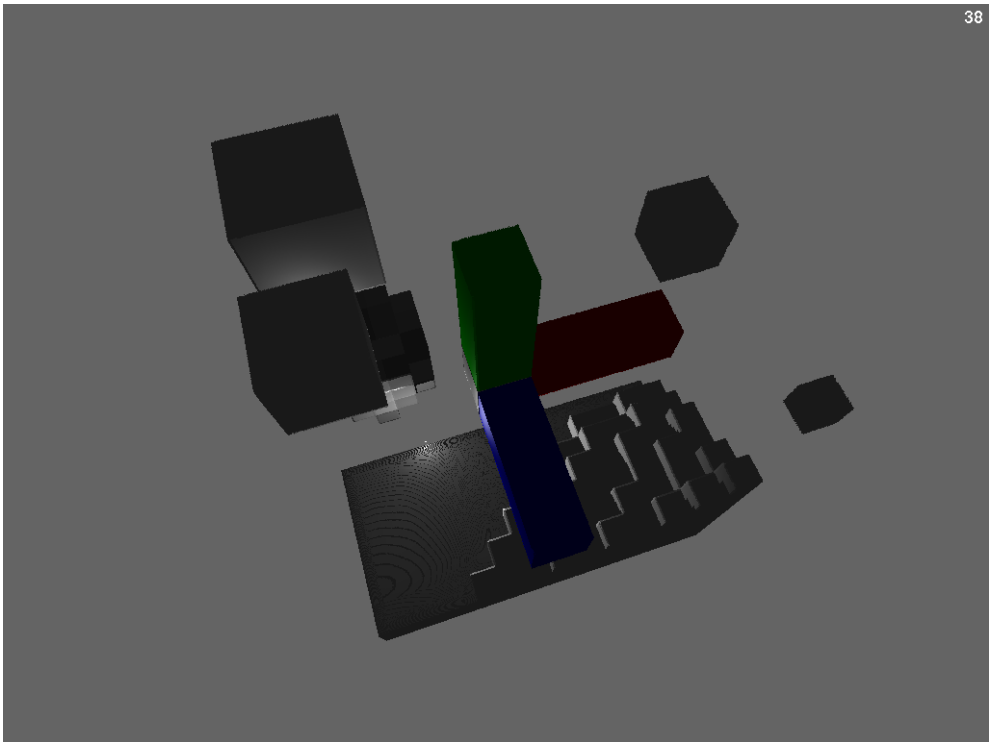
Lineaarse ja lähima filtreerimise võrdlus (kaugusfunktsiooniga kiirendatud, üks joonistuskord)



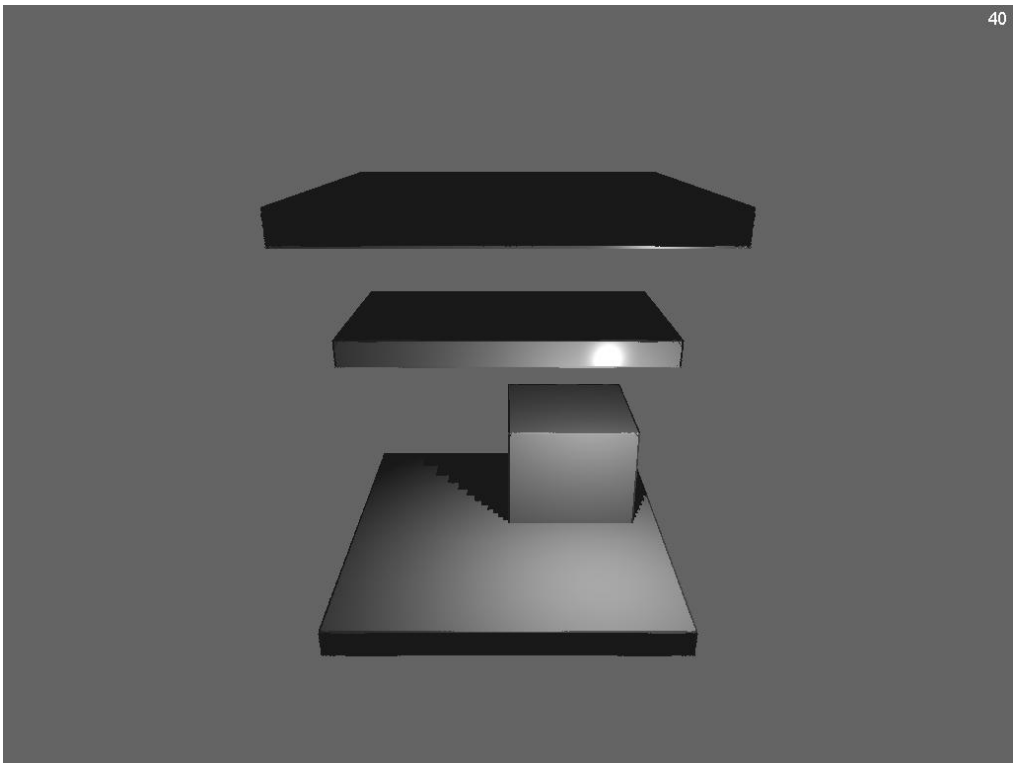
Kaks valgusallikat (kaugusfunktsiooniga kiirendatud, üks joonistuskord, sisendi silumine)



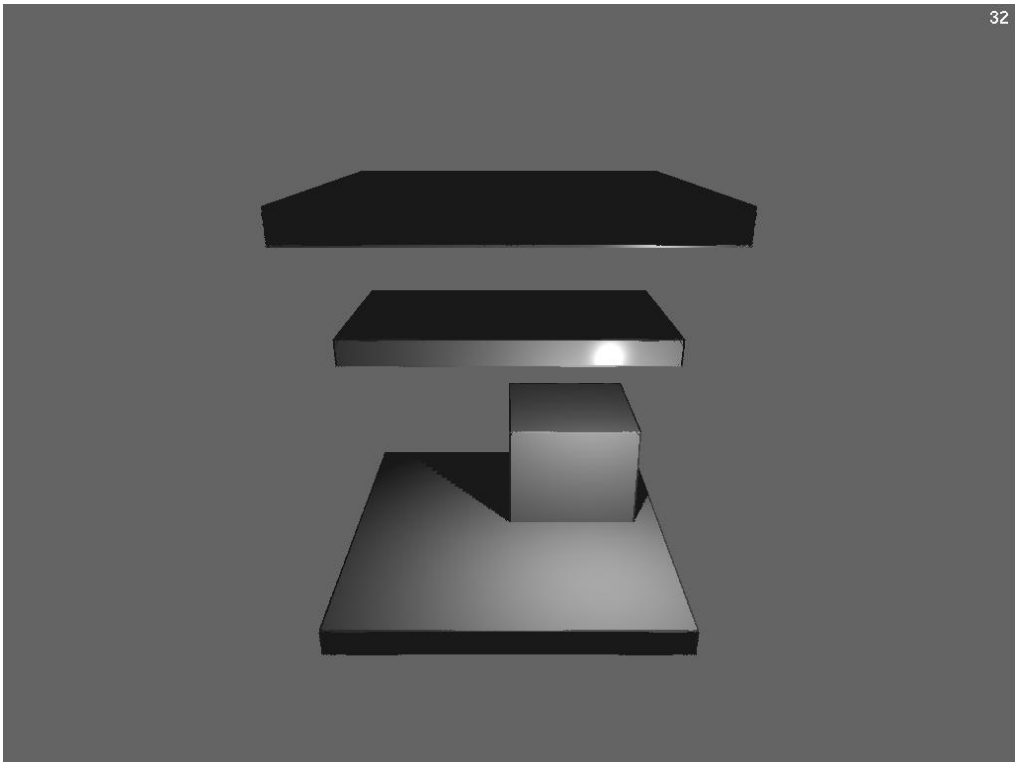
Peegeldused (kaugusfunktsiooniga kiirendatud, üks joonistuskord, sisendi silumine)



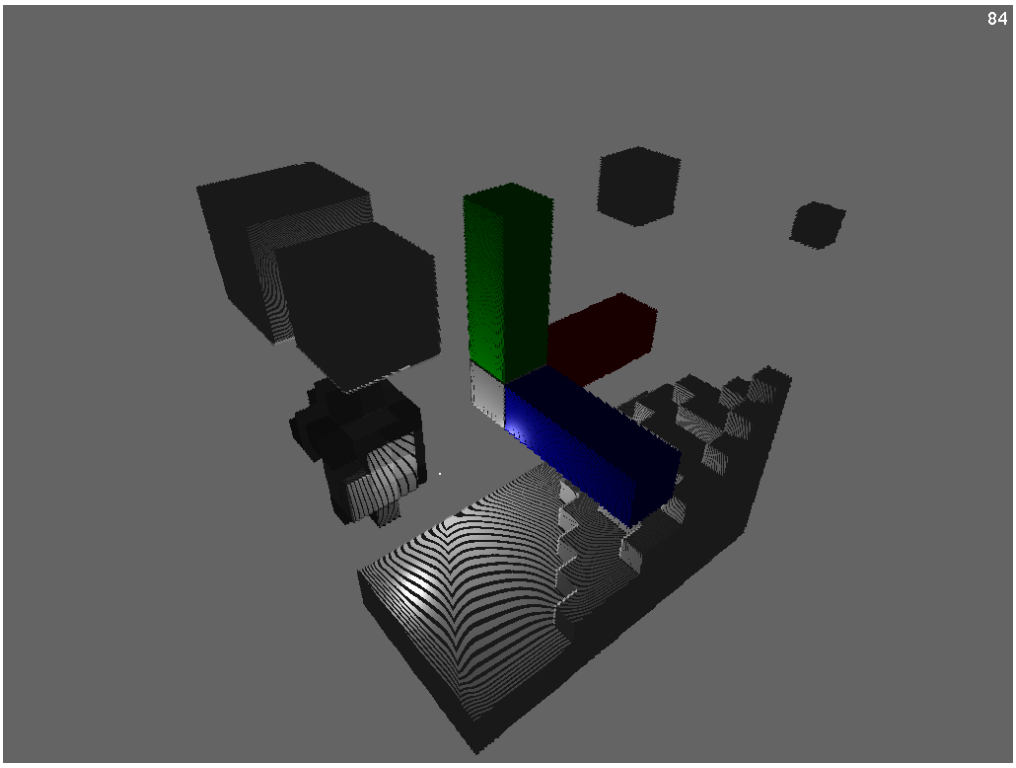
Defekt varju arvutamisel pinnalähedase punkti nihke arvutamise tõttu



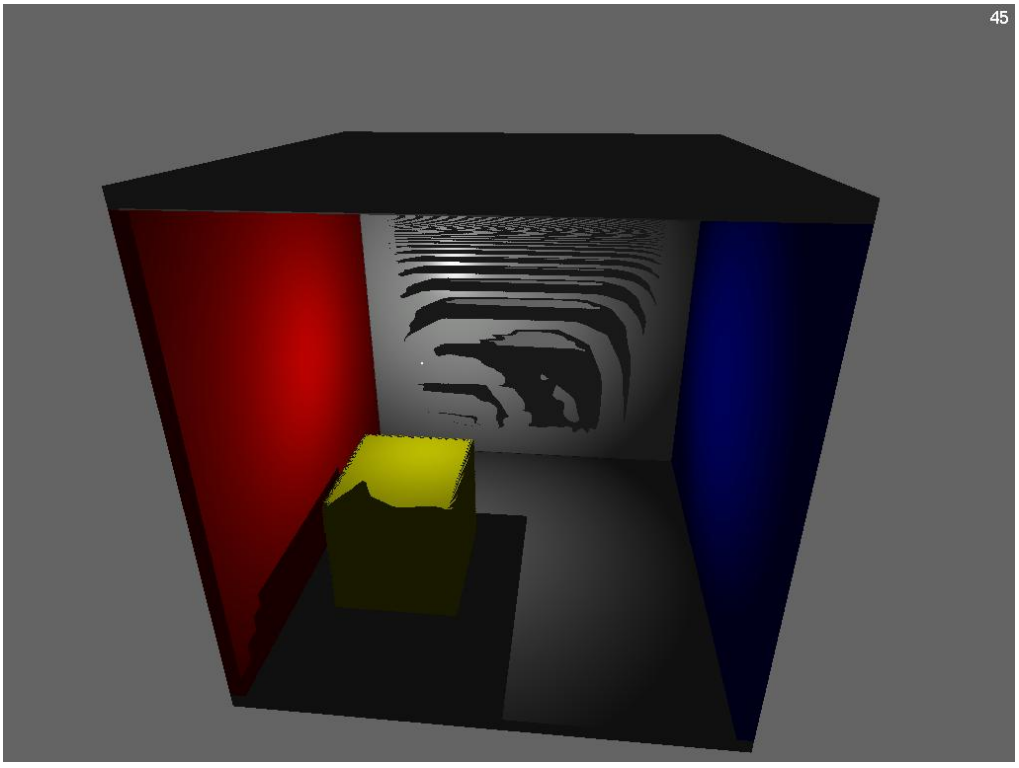
Defekt varjude arvutamisel konstantse sammude arvuga (20 sammu)



Defekt varjude arvutamisel konstantse sammude arvuga (50 sammu)



Defekt liiga väiksest sammude arvust kiire jälgimisel



Defekt liiga suure sammu pikkuse kordaja valimisel kiire jälgimisel